A Preliminary Architectural Design For The Functional Hierarchy Of The INFOPLEX Database Computer

Meichun Hsu

Technical Report # 5
WP 1197-81
November 1980

Contract Number N00 39-80-K-0498
Internal Report Number M010-8011-05

A Preliminary Architectural Design For The Functional Hierarchy Of The INFOPLEX Database Computer

Meichun Hsu

Technical Report # 5
WP 1197-81
November 1980

Principal Investigator:
Professor S.E. Madnick

Prepared for:
Naval Electronics Systems Command
Washington, D.C.

ABSTRACT

Conventional computer architecture has shown its limitations in supporting large-scale information processing. Database machines which specialize in database functions have been suggested to alleviate the problem of increasing loads on these computers.

The INFOPLEX database machine proposed by Madnick employs a highly-parallel computer architecture to achieve high performance and capacity. It contains two subsystems: the storage hierarchy and the functional hierarchy. This paper is about a design of the functional hierarchy, the subsystem of the INFOPLEX which performs database functions.

As originally propsosed by Madnick, the functional hierarchy is made up of hierarchical levels; each level is designed to perform certain databasae functions and is to be implemented by multiple microprocessors. The guidelines for identifying these levels are (1) pipelining of transactions, and (2) functional abstraction. This paper attempts to turn this original idea into a detailed design by accomplishing the following: (1) Identification of functional requirements (i.e., features) of a generalized DBMS; these features are to be supported by the functional hierarchy; (2) Development of an integrated data model; (3) Detailed specifications of architectural levels, including their functions and implementation strategies; and (4) Pointing out future research directions.

TABLE OF CONTENTS

I. INTRODUCTION

Conventional computer architecture has shown its limitations in supporting high-performance, high-reliability and large-capacity systems dictated by today's information processing needs. Attempts in the form of microcoded instruction sets, intelligent controllers, back-end processors and database machines have been made to augment the data processing capability of a computer <Hsiao77>. The INFOPLEX database computer represents one effort which employs a highly parallel computer architecture designed specifically for such information procesing needs.

Using the concept of hierarchical decomposition in its design, multiple microprocessors in its implementation, and decentralization in its control mechanism, the INFOPLEX database computer architecture has as its objective the support of large-scale information management with high reliability <Madnick79>. It aims to provide a solution to the problem of increasing loads, in terms of both throughput and volume of stored data, faced by today's and tomorrow's information processing nodes.

INFOPLEX consists of a storage hierarchy, which supports a very large data storage system, and a functional hierarchy, which is responsible for providing all database management functions other than device management. The functional hierarchy is built on top of a storage hierarchy. This data computer may be used as a stand alone database machine, where users interact with it directly through a Data-Definition/Data-Manipulation/Query language interface, or it can

be connected to a regular host computer through a data channel, where the host computer augments the database machine's functions by providing language processors and other utilities. The host computer generates commands to be received by INFOPLEX's DDL/DML/Query interface, as depicted in Fig 1.1.

1.1 Storage Hierarchy

The storage hierarchy is comprised of levels of storage devices with various performance and cost features. In our research <Madnick75>, it is found that the requirements of a high-performance and low-cost storage system are best satisfied by a mixture of technologies combining expensive high-performance devices with inexpensive low-performance devices. Hierarchical decomposition is applied to organize this ensemble as a hierarchy. The high-performance devices, such as cache memory and main memory, are placed on the top (i.e., the highest level of the hierarchy), while low-performance devices such as mass storage systems are placed at the bottom (i.e., the lowest level of the hierarchy). An example of our storage hierarchy is shown in Fig 1.2.

The storage hierarchy supports the functional hierarchy by providing a very large linear virtual address space with a small access time. The actual structure of the storage hierarchy and movement of information between levels within the storage system are hidden from the functional hierarchy. The lowest level of the storage hierarchy always contains all the information of the system, while higher levels

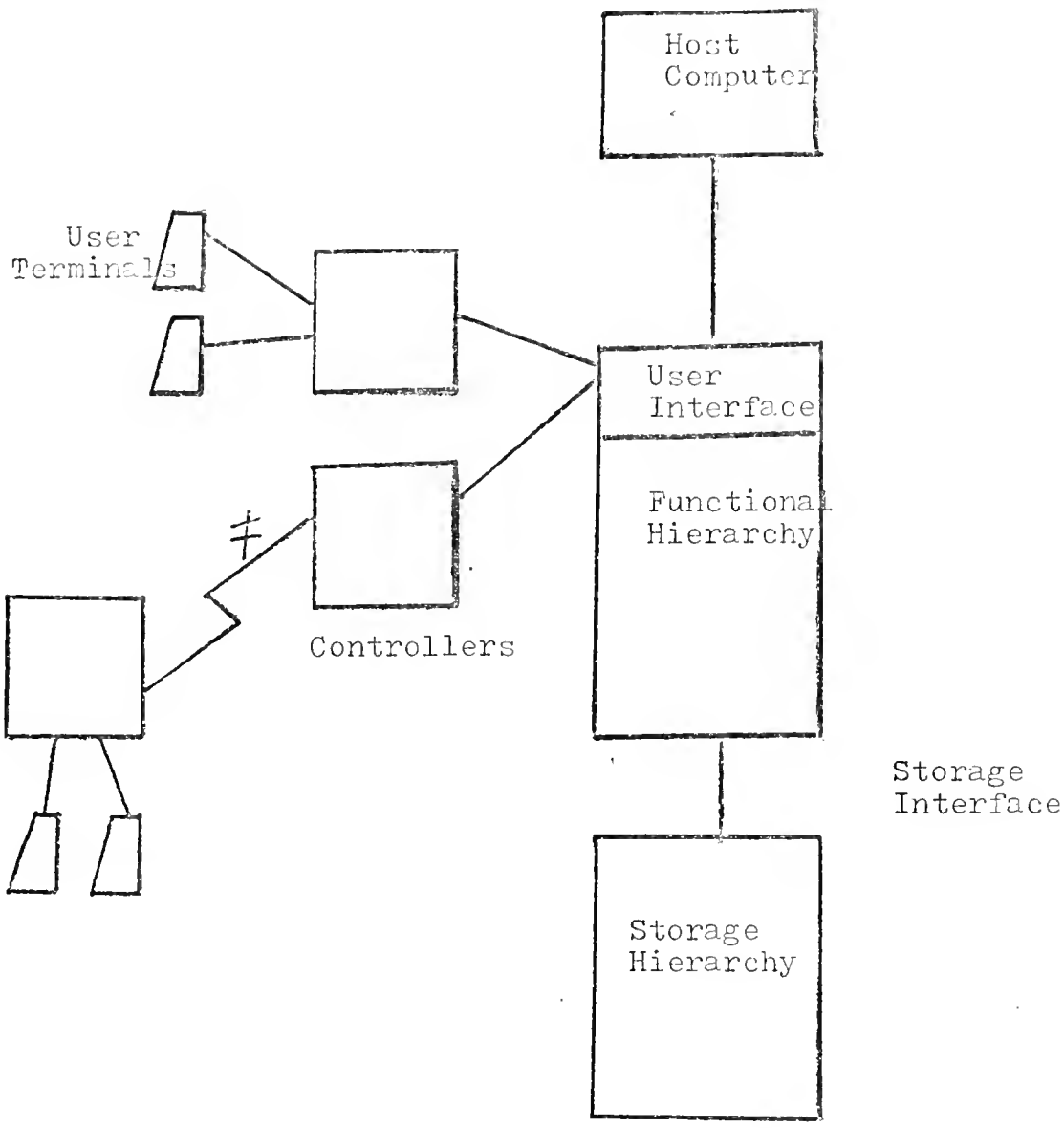


Fig 1.1 INFOPLEX: Backend machine/Stand-alone data computer

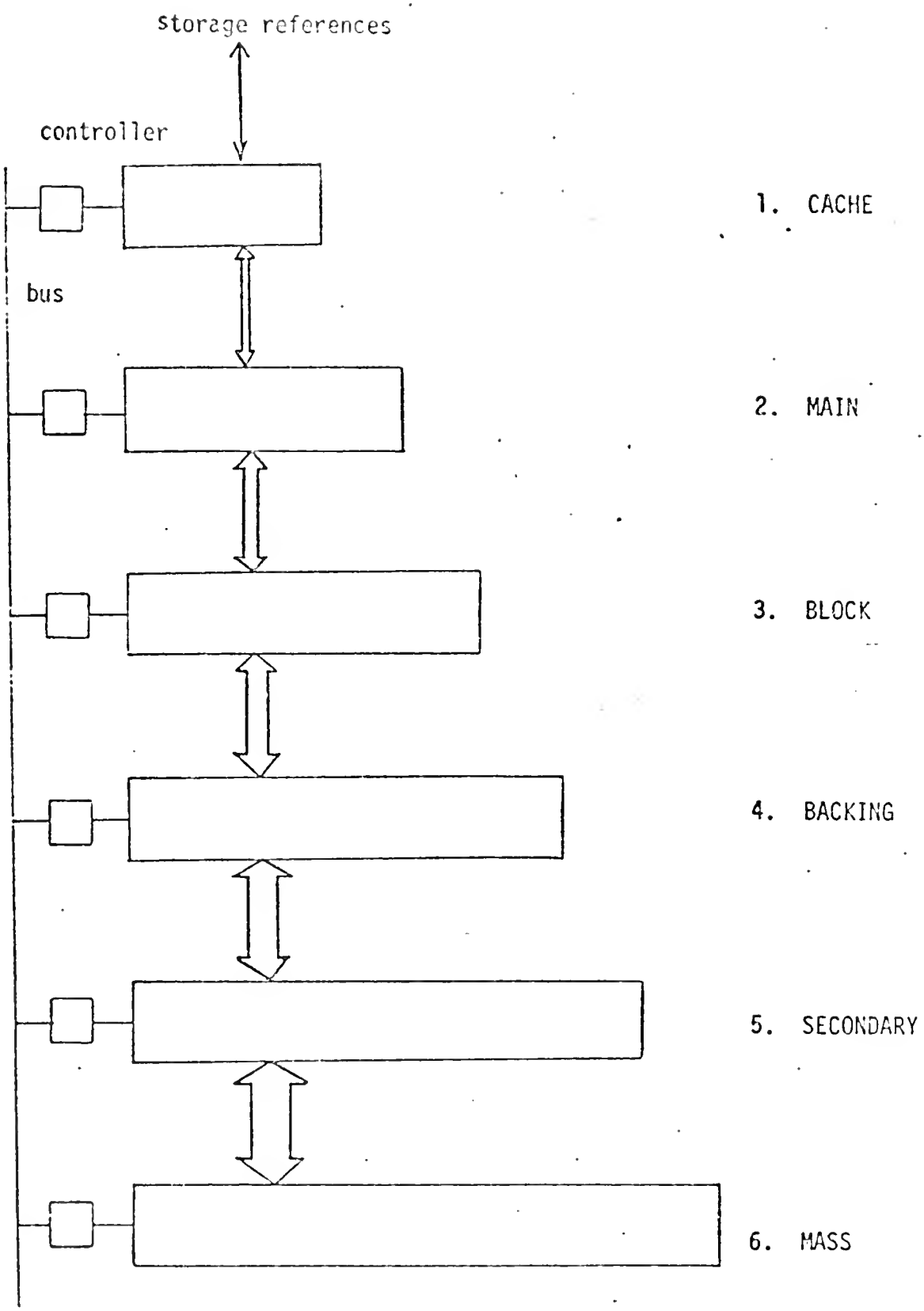


Figure 1.2 An Example Memory Hierarchy

contain subsets of the total database. Requests for data are made to the highest level, and information is moved automatically between levels depending upon actual or anticipated usage, such that the information most likely to be referenced in the future is kept at the highest level. The effectiveness of the storage hierarchy therefore depends heavily on locality of reference.

Microprocessors are used at each level to implement data movement algorithms. Simultaneous and parallel operations at all storage levels enhance throughput and reliability of the storage system. Various desirable properties of storage hierarchies have been identified, and the relationships between these properties and various storage management strategies have been studied in detail <Lam79, Abraham79>.

1.2 Functional Hierarchy

1.2.1 Hierarchical functional decomposition

The INFOPLEX functional Hierarchy is designed around a concept of hierarchical functional decomposition. The concept of hierarchical decomposition, as applied to the functional design area, is a technique that identifies the key functional modules that have minimal interdependencies and can be combined hierarchically to form a software system, such as an operating system or a database management system.

1.2.1.1 Hierarchical vs. non-hierarchical design

The advantages of hierarchical modular design as opposed to conventional subroutine modular design can be shown by a simple comparison. Fig 1.3 depicts a system consisting of a "main" program and eight subroutines. In addition, there is a common data pool used by all of these routines. Each link in the figure represents an interdependency. If the function or interface to suroutine E were changed, five other subroutines (i.e. M, A, B,C, and D) may also have to be changed. The same argument applies to changes in the format or usage of a variable in the data pool. In the hierarchical decomposition approach, functionality is distributed to modules in a very strict manner so as to produce a hierarchical structure as illustrated in Fig 1.4. In this case, the modules are designed such that each of the nine routines is only dependent upon one other routine. Furthermore, each routine maintains its own private data pool as needed to serve its function. In such a case, if a subroutine is modified, there is only one other subroutine that can be directly affected and must be tested. Similarly, a change in a data pool variable only impacts a single subroutine. Besides minimizing the propagation of changes, this hierarchical approach also makes it much easier to determine which subroutine must be changed, and in what manner, since the functionality of each subroutine must be well-defined. This approach has been found to be very effective in earlier design work on file systems <Madnick69, Madnick74>.

Another reason for the choice of a hierarchical design is to take advantage of the pipelining nature of transaction processing in a database system. A transaction that enters a database system normally has to go through a sequence of stages of processing. For example, it

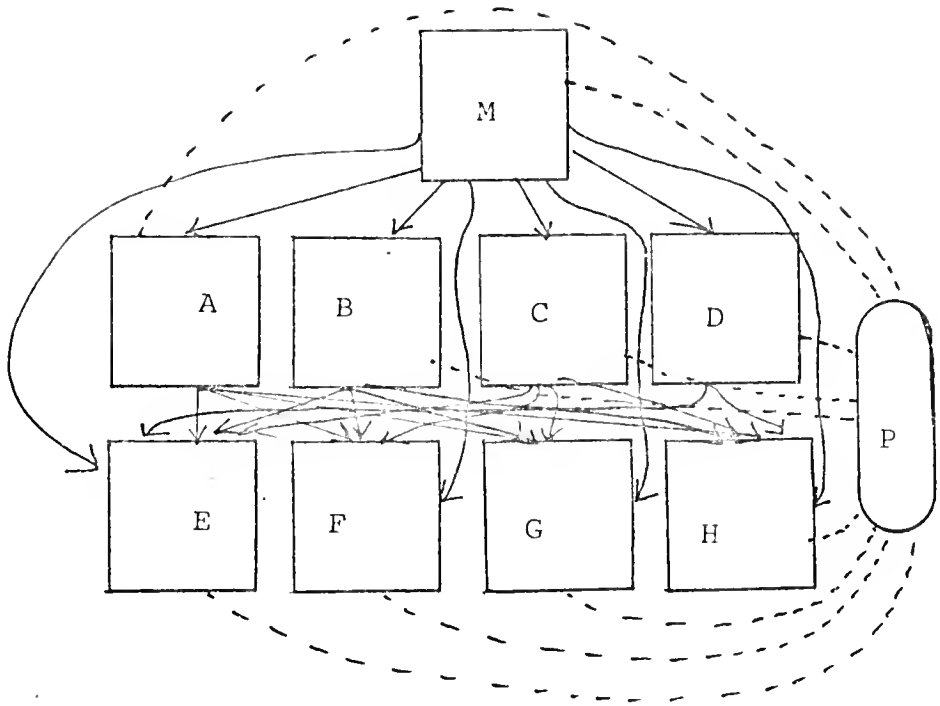


Fig 1.3: Non-hierarchical design

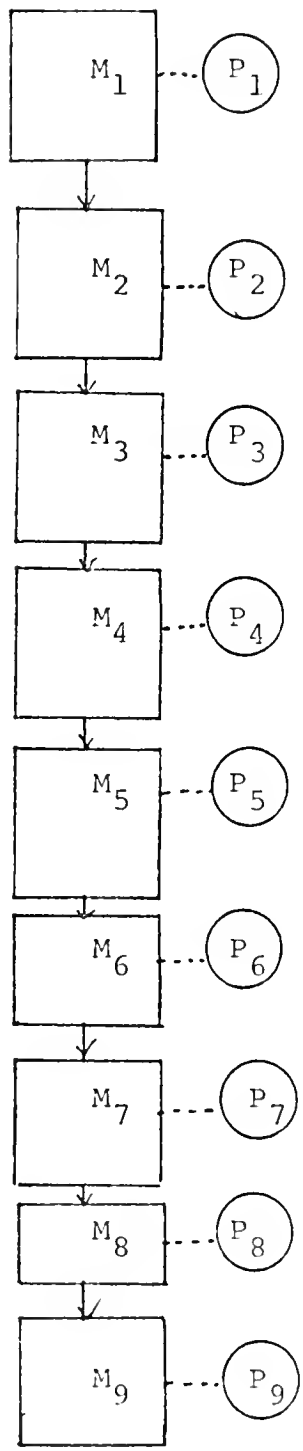


Fig 1.4 Hierarchical design

may first be checked by a security control module; then it is passed to a name-mapping module which determines the records to be accessed; and then it is given to a search module which determines the address of the records; finally a storage module is invoked to obtain the record from the memory. These stages suggest strongly a database system structure that reflects their sequence. Moreover, the modules that support the earlier stages of processing (e.g., security control and name-mapping) also require services provided by those modules that support the later stages of processing (e.g., searching and accessing). Research in stratification of database systems will be reviewed in chapter 2. Here we conclude by pointing out that, since hierarchical modularity enables higher-level modules to be implemented on top of an 'extended machine' incorporating all the primitives implemented at lower level modules, it contributes to the reduction of redundancy of functions in the system, and therefore enhances the reliability.

1.2.1.2 Family of systems

Another advantage of the decomposition approach lies in the development of a 'family of systems'. The concept of 'family of systems' is motivated by the proliferation of operating systems or DBMS's developed over the past decade or two. While the individual systems may have various desirable features, it is often difficult for the user to find a system which possesses just the right number of qualities that he desires. It is therefore advantageous to develop a general structure that can be used as a basis for many different systems. By decomposing a system into modules that are compatible through a set of well-defined interfaces, it is nearly possible to

develop any specific system by an appropriate choice of modules proposed. In other words, a desired system may be assembled according to needs the way a stereo system or a cutomized automobile is assembled. In our research on Family of Operating Systems <FOS76> and Family of Database Management Systems <FODS76>, hierarchical modular design was found to be effective in providing a normative model for the systems.

Members of a family of systems will differ as a result of differences in the contents of the modules that make up the hierarchy. There are three broad classes of module differences:

(1) Functional: Although the purpose of and the interface to each module must be clearly defined, the specific functions and algorithms used may vary significantly.

(2) Performance: For a given functionality, there may be different implementations that offer different performance characteristics.

(3) Existence: As an extreme case of minimal functionality, certain modules may not exist at all in certain system.

Applying the concept of heirarchical decomposition, the INFOPLEX Functional Hierarchy attempts to decompose the typical DBMS functions, such as data restructuring, security control, integrity and validity checking, access path optimization, data encoding, etc., into a hierarchy of tasks, each of which to be implemented as a 'level' of the hierarchy. Levels are connected in a top-down fashion, with higher level modules supported by the 'primitives' of the 'extended machine'

composed of the hardware and all lower level functional modules. However, design and implementation of each level is made as independent of other levels as possible so that algorithms incorporated in a particular level are not affected by those of other levels. In particular, inter-level communication is made through a set of clean, pre-defined interfaces.

1.2.2 Multiple Microprocessor Implementation

The functional levels specified above are to be implemented using multiple microprocessors to take advantage of possible parallelism and pipelining effects in processing incoming streams of transactions. As shown in Fig 1.6, each level in the hierarchy communicates only with adjacent levels and each module within a level communicates only with adjacent modules. Thus no central control mechanism is necessary.

When a processor in a level requires service from the next lower level, it places an operation code and associated operands in a shared memory area accessible to only these two levels. This special memory module is called an Interlevel-Request-Queue, or IRQ, to be distinguished from the storage hierarchy and the local memory described below.

In addition to the IRQ, every level has some local memory as working space. Therefore, each processor may have 3 sets of memory modules:

(1) IRQ shared by the next higher level,

(2) local working space, and

(3) IRQ shared by the next lower level.

This concept is shown in Fig 1.6.

Even though a processor has a number of memory modules, memory operations can be supported in a conceptually simple fashion by assigning different ranges of the address space of the processor to each of its three memory modules. Thus the same set of storage operations (i.e.LOAD,STORE,MOVE,etc.), and the same addressing mechanisms can be used for each of the three types of memory.

To illustrate, suppose the data encoding level (refer to fig 1.5) places a request to the memory management level to fetch a byte string of data, given its 'id'. (The 'id' is a unique identifier for the byte string, and is described in detail in section 2.1) The calling module formats a message and stores it in the IRQ shared by the data encoding and memory management levels. This message contains an operation code (i.e. 'FETCH') and the id of the data element to be retreived. When the memory management level completes this request, it stores the byte string of data in the IRQ, and returns a message to the data encoding level, containing a pointer to the data in the shared IRQ.

There are several ways to implement this hierarchical ensemble of processors and memory modules. One approach is to simulate the hierarchical structure of the system with a linear, single bus, structure (Fig 1.7). Research in the area of multiple microprocessor networks has shown that improved communication protocols and bus architectures can be used, with today's technology, to linearly connect

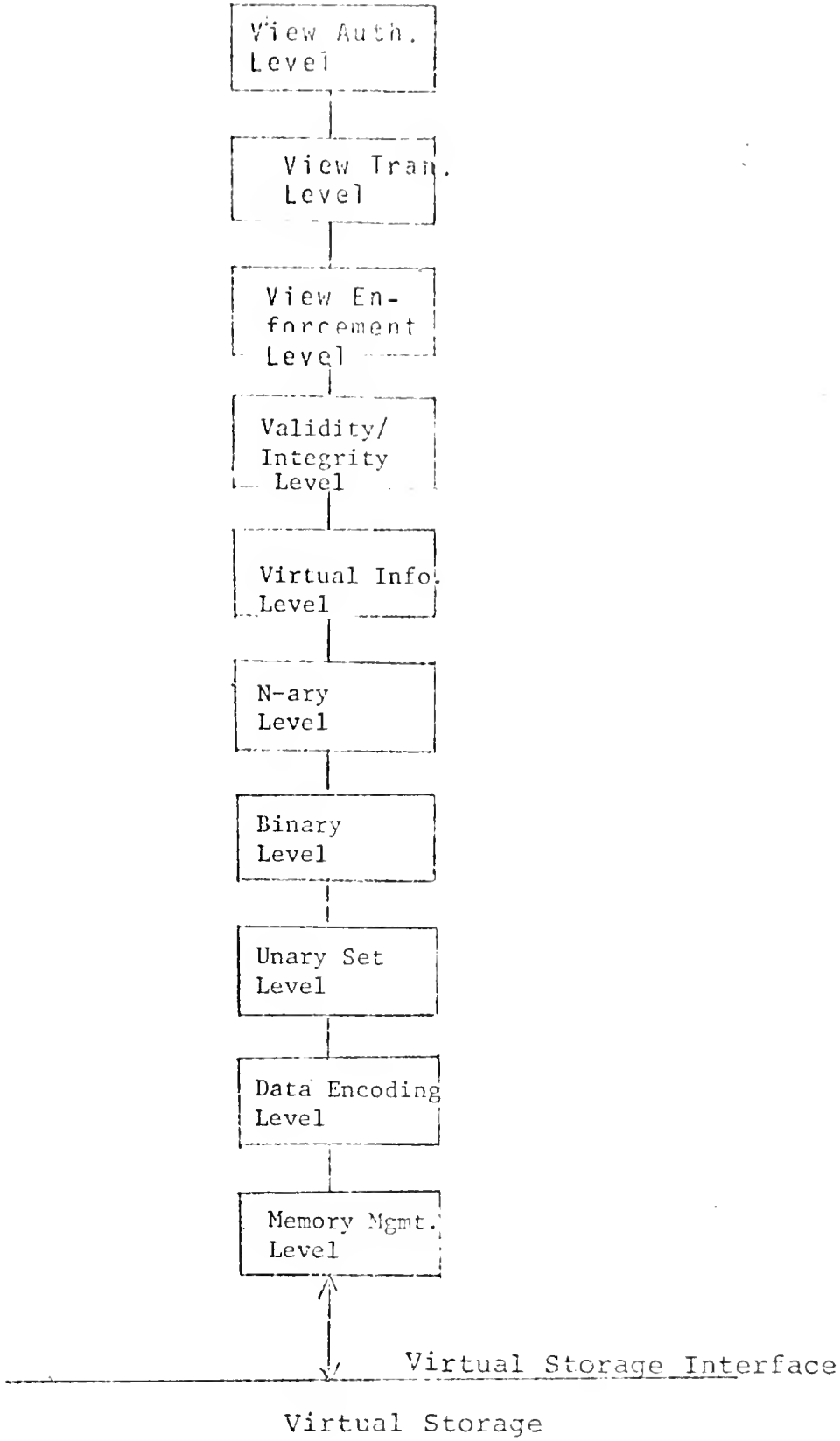


Fig 1.5: Functional Hierarchy

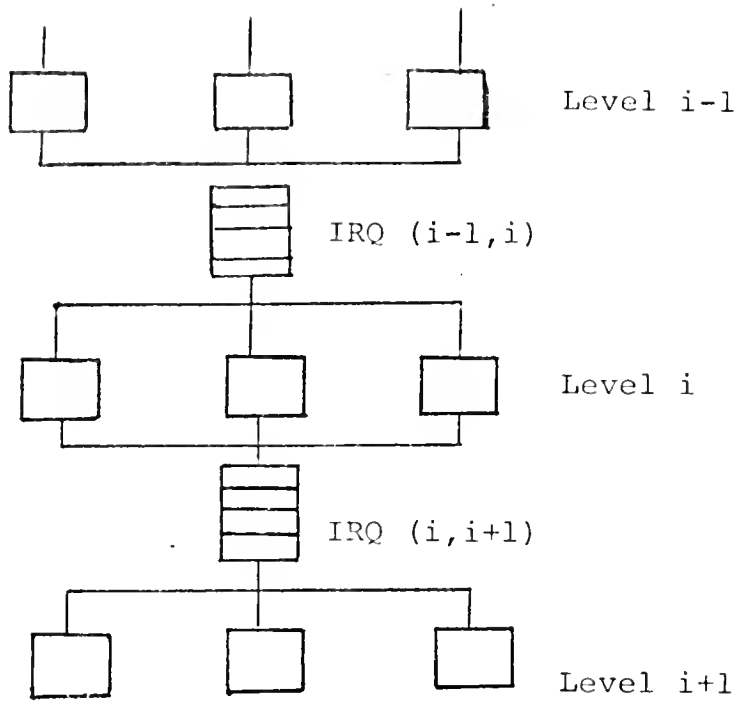


Fig 1.6 Multi-processor implementation of Functional Hierarchy

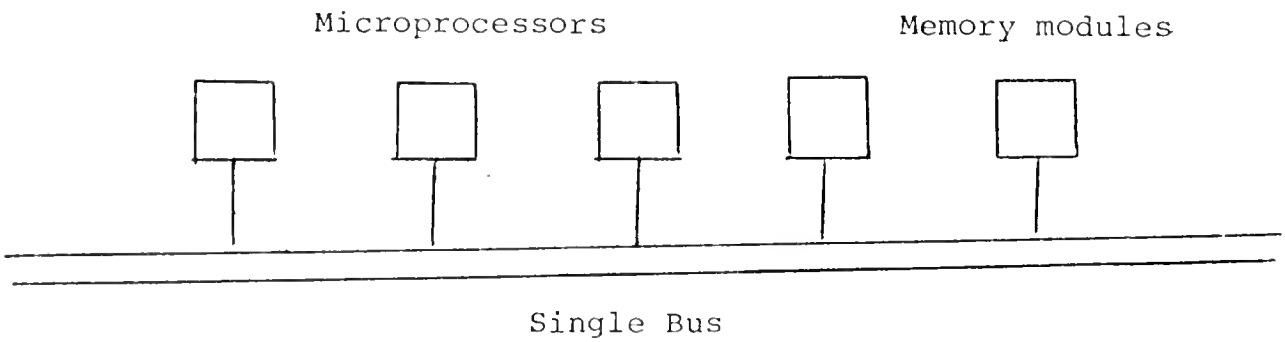


Fig 1.7 Simulating Functional Hierarchy with a single-bus microprocessor ensemble

up to 60 microprocessors without any performance degradation due to communication bottlenecks on the bus<Toong80>. Another approach, taking advantage of new fabrication technologies, is to implement each functional level using several multi-microprocessor multi-memory clusters. As illustrated in Fig 1.8, each of these clusters is referred to as a functional processor cluster (FPC), and can be fabricated on a single chip. The IRQ may also be implemented using a FPC, whose data buses communicate with adjacent functional levels, as shown in Fig 1.9 <Madnick80>.

1.2.3 Summary

The advantages of the functional decomposition approach to database computer design are summarized below:

(a) Decomposed design and implementation: Functional decomposition breaks the design and implementation of a potentially very large DBMS into smaller, much-easier-to-tackle modules, where each module can be worked on separately.

(b) Modularity: Each level of the functional hierarchy interacts with other levels through a clean set of interfaces; therefore modules that perform the same task while using different algorithms or employing different levels of sophistication can be selectively plugged into the system.

(c) Parallelism and pipelining: Multiple microprocessor implementation makes it possible to process very high transaction rates (orders of magnitude higher than currently available systems).

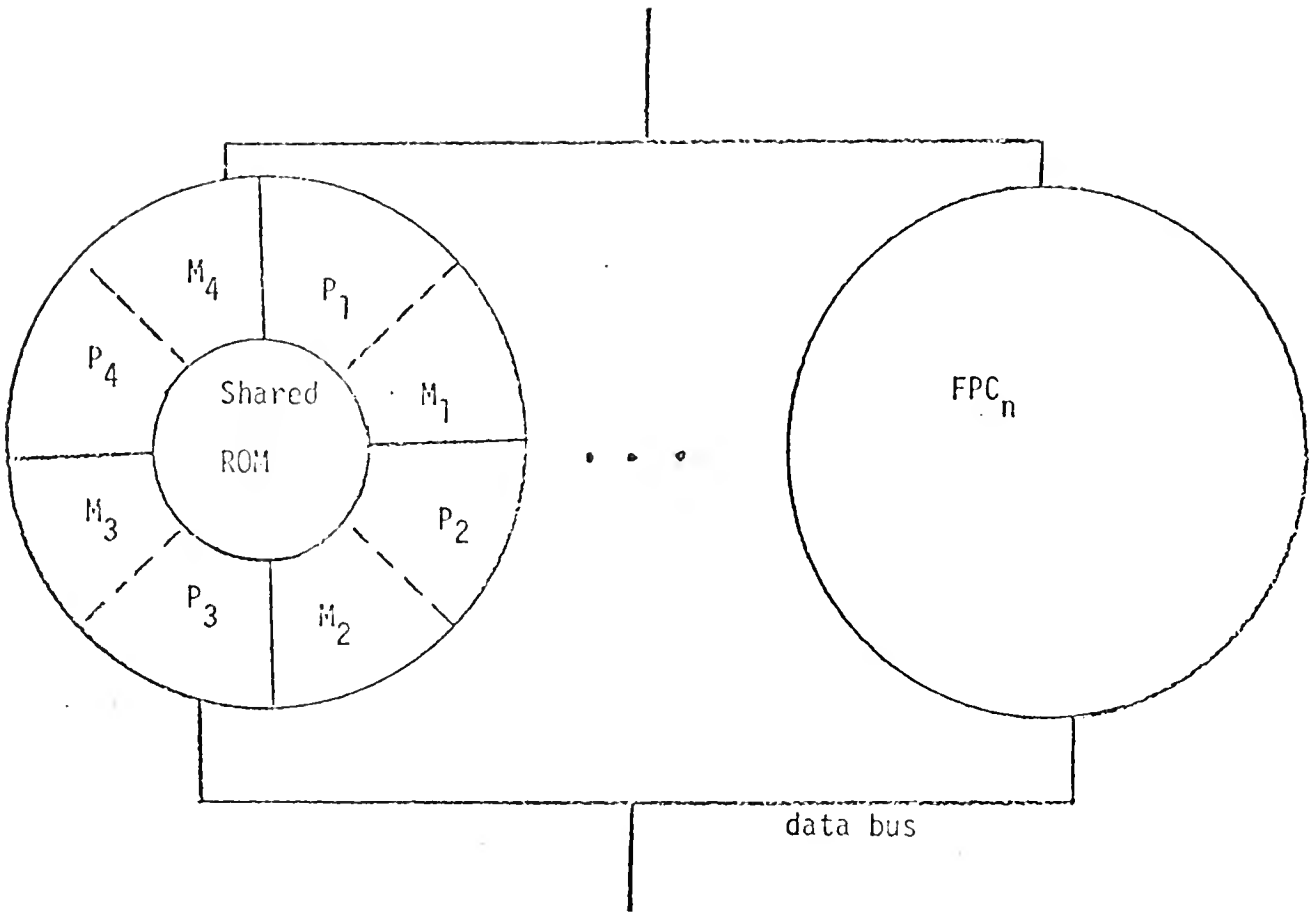


Fig 1.8: Functional Processor Clusters (FPC)
in a functional level

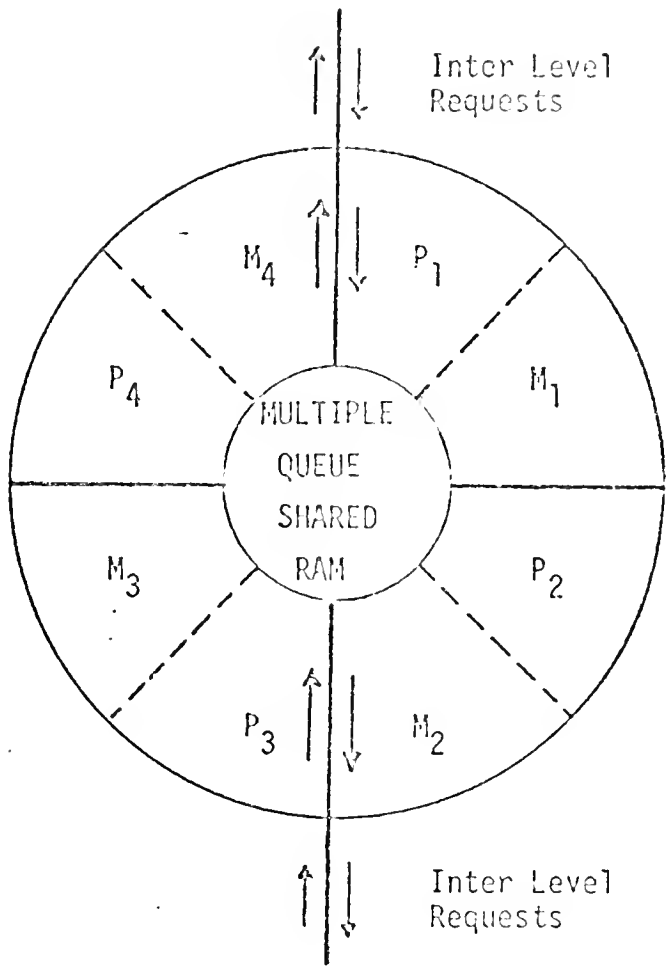


FIGURE 1.9

IRQ Structure as a Functional Processor Cluster (FCP)

(d) Distributed control: Since each module has clearly defined functions, it is easier to detect errors and to identify the erroneous module. Also the use of multiple parallel processors at each level enhances the availability of the system in the event of any isolated software or hardware breakdown.

1.3 Research goals and an overview of this report

The goal of this research is to turn the INFOPLEX concept of a pipelined, multi-processor-based database management system into a detailed design. Specific accomplishments are the following:

(1) Identification of functional objectives of the system: We have identified, drawing from the current literature in the DBMS area, the following as major features to be included in the design of the system:

(a) multiple types external views of the database

(b) a high-level conceptual data model rich in semantics

(c) a flexible physical data structure

(d) explicit support of database security, validity, alerting constraints and virtual information

(e) concurrent use of the database

(2) Development of an integrated data model: we have developed a data model, which is used to describe the database and serve as a media for inter-level communications

(3) Specification of architectural levels: We have examined database stratifications in the past and proposed a more

generalized layered architecture to achieve our functional objectives. In particular, functions performed and data structures used to implement them at each level are described. This can be used as a blue-print for software prototype implementation and for future study and refinement.

In this chapter, we have reviewed the architecture of the INFOPLEX database machine and introduced basic design concepts of the functional hierarchy as outlined in <Madnick79>.

In chapter two, the general structure of the functional hierarchy is discussed. It describes the rationale for the proposed stratification in an integrated fashion, and relates it to the literature of various database research areas.

The rest of the report is organized around the proposed structure of the Functional Hierarchy. For each level identified in the Functional Hierarchy, the following general issues are discussed:

1) The functions this level performs;
2) The rationale for singling out this level;
3) The implementation strategies;
4) The interfaces;

As shown in Fig 1.5 , our design of the INFOPLEX Functional Hierarchy has the following levels:

A. Memory Management

1) memory management level

B. Internal Structure

2) data encoding level

3) unary set level

4) binary association level

C. Database Semantics

5) n-ary entity level

6) virtual information level

7) data validity and data integrity level

D. User Views and Database Security

8) view enforcement level

9) view translation level

10) view authorization level

The report starts from the lowest level of the Functional Hierarchy. In chapter three, we discuss the memory manager. The discussion focuses on how this level is to be implemented in order to manage the virtual memory resource and to provide mapping functions of the logical identifier of a data element to its physical identifier in the virtual memory.

In chapter four, we describe how the internal structures of the database are supported. Descriptions of the three levels supporting the internal structure, namely, the data encoding level, the unary set level, and the binary association level are presented.

The encoding scheme of a stored data element may change when the

element is passed from one level to another. In particular, an element may go through data compaction, editing, or various forms of encoding right before it is to be stored into the storage hierarchy. The data encoding level provides functions to perform these types of data conversion.

Stored data elements are grouped into unary sets. (The notion is similar to that of grouping stored records into files.) The unary set level deals with search and retrieval of stored data elements from the unary sets. It provides a "content-addressable" interface to its superior levels. It incorporates data structures that facilitate searching into the database.

The binary association level implements binary connections specified in the conceptual schema. Even though it is classified as one of the internal structure levels, it provides the basic service to materialize complex semantic constructs. It is capable of extracting a data element from the database given the content of an associated element.

In chapter five, we discuss how database semantics may be built in and maintained. A structure of 3 hierarchical levels is proposed. At the n-ary entity level, binary associations are grouped into an n-ary construct that is used to describe an entity in the real world. Multi-valued attributes as well as nested attributes are built into this n-ary construct. This level supports an n-ary entity interface, which returns an entity based on some description of the attributes of this entity. It also has to resolve certain access path selection

problems.

The virtual information level provides functions to derive information which is not physically stored. The validity control level further implements constraints on updating of the database. These two levels complete the discussion of database semantics.

In chapter six, we describe the implementation of definitions and mappings of external views and control of database security. A three-level functional structure is discussed. The view translation level sits in the middle, performing mapping and translation of views. Three kinds of views are discussed: relational views, hierarchical views, and network views. It shows, by way of a sample database, how different views and their operators may be translated.

Below the view translation level, the view enforcement level integrates all external views and enforces operational access constraints. On the other hand, the view authorization level on top of the view translation level authentiates log-on users and authorizes views to the users.

Finally, in chapter seven, we conclude this report, and point out dimensions for further research in the design of the functional hierarchy.

II. The General Structure of the Functional Hierarchy

Before we go on to the description of individual levels in the hierarchy, an integrated overview of the stratification proposed is presented here. This chapter dwells on the recent literature in the area of database design and database management systems, and its relationship to the design of the Functional Hierarchy.

2.1 Stratification of the DatabaSe Management System

In this section, we shall review two important concepts in stratification of database management systems. The first one, represented by the ANSI/SPARC recommendations, emphasizes the process of data abstraction and a three-level hierarchy of data models. The other one, represented by the DIAM model, stresses abstraction of functions. Both have been drawn upon for determining features to be supported by the functional hierarchy and its architecture.

2.1.1 The ANSI/SPARC DBMS architecture

It is one of the objectives of the INFOPLEX Functional Hierarchy to be able to support various high level constructs demanded by an information modeller. In order to design a DBMS that has the capability to provide many different kinds of views (e.g. relational, hierarchical or network views) of the database, as well as the flexibility in the organization and reorganization of the stored data, INFOPLEX has adopted a DBMS architecture similar to that suggested by the ANSI/SPARC study group <ANSI75, Yourmark77, Tsichritz78>. Under

this framework, as shown in Fig 2.1, a conceptual schema is introduced to insulate view definitions (i.e. the external schema) from stored structure definitions (i.e. the internal schema). The application program views are mapped to the conceptual schema, such that changes or additions of individual views will not affect the definitions of the others. On the other hand, conceptual schema is mapped to the internal structure, such that changes to the internal structure will affect only the mapping between the conceptual schema and the internal schema, but not the external views. Therefore data independence may be preserved and protection of existing application programs can be effected. To serve its purpose, the conceptual schema should have the following properties:

(1) It is a description of the enterprise that will stay relatively stable compared to the external or internal schema.
(2) It is capable of expressing high level semantic constructs existent in the enterprise in order to faithfully model the enterprise.
(3) It is simple to work with and flexible in restructuring itself to provide different external views.

A very similar architecture is found in the description of System R <Astranhan76>. As shown in Fig 2.2, System R has a Relational Storage System (RSS) which corresponds to the internal level, a Relational Data System (RDS) which corresponds to the coceptual level, and various programs run on top of the RDI to support other user interface.

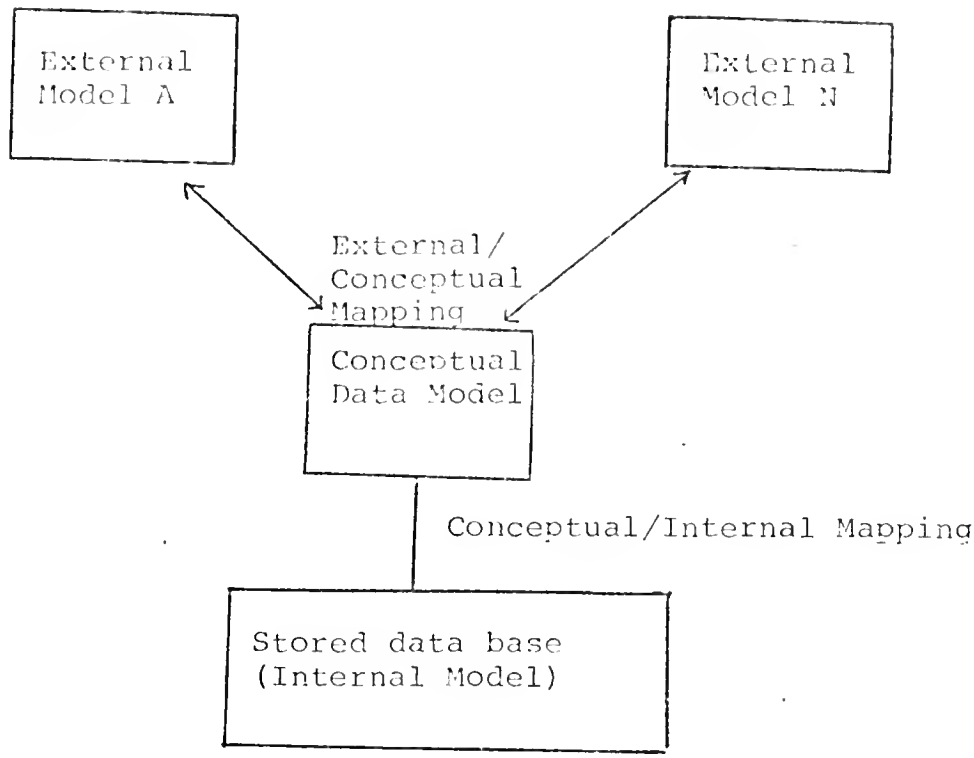


Fig 2.1: The ANSI/SPARC DBMS Architecture

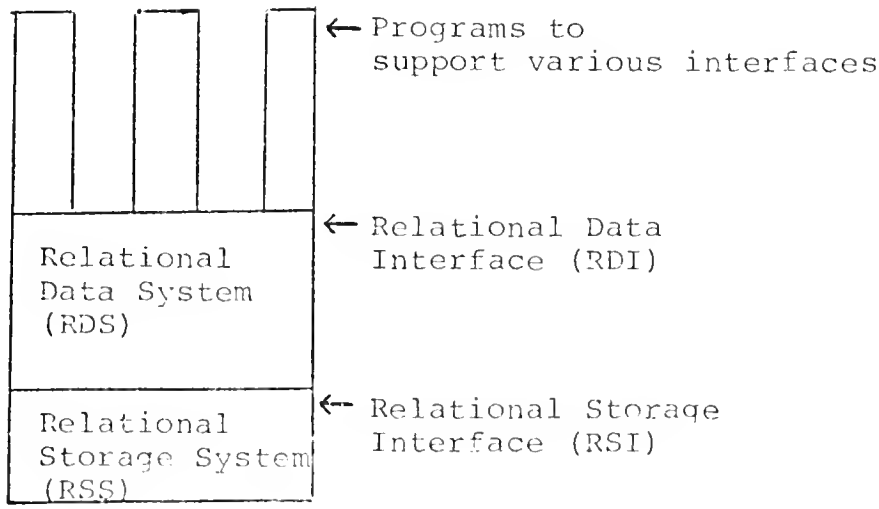


Fig 2.2: The System R Architecture

The choice of data models has a great impact on the design of interfaces between levels of the functional hierarchy. We shall examine, in section 2.2, the significance of choices of data models at each of the three levels, and approaches taken in the design of the Functioanl Hierarchy.

2.1.2 The DIAM concepts

Senko <Senko73> has also exploited to a great extent the concept of stratification in implementing a database system. His Data Independent Accessing Model (DIAM), as shown in Fig 2.3, has identified four levels of abstraction for a DBMS: the Entity Set model (the info-logical level), the string model (the construct-building level), the Encoding model (the basic construct implementation level), and the physical device model. Even though there is certainly a similarity between DIAM and the ANSI/SPARC architecture, the purpose of DIAM's stratification is more along the line of abstraction of database functions. This results in a further decomposition of its internal model, with the emphasis that a higher layer always builds its functions on top of those implemented at a lower layer. Another example of stratification of database functions is presented in <Navathe76> in a more limited context.

2.1.3 The INFOPLEX Approach

The architecture of the Functional Hierarchy strives to achieve the following features of a DBMS:

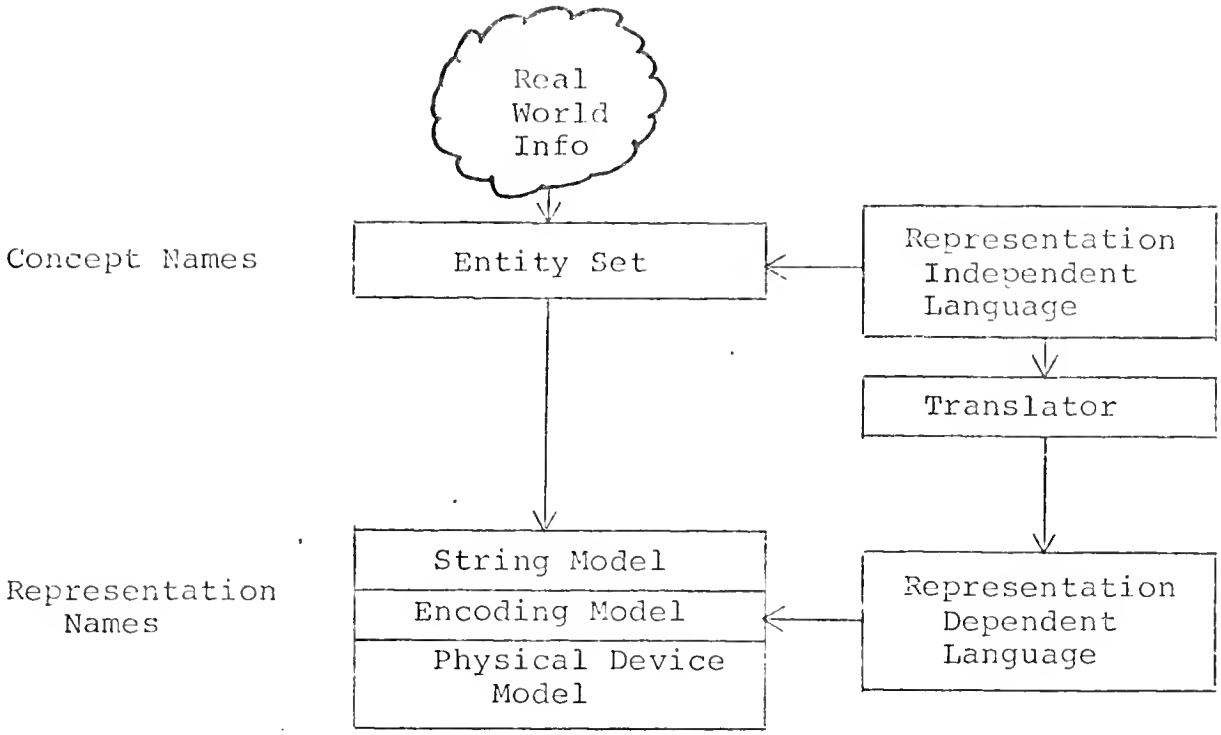


Fig 2.3: The DIAM Architecture

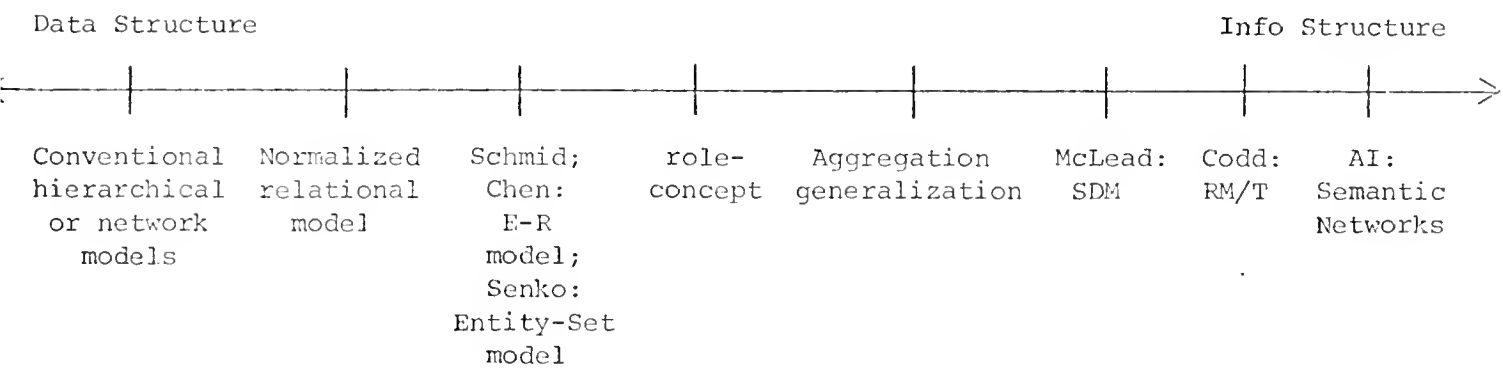


Fig 2.4: Conceptual Data Modeling

(1) Support of multiple types of views

(2) Separation of the structure of views (the 'external schema') and the structure of the stored data (the 'internal schema') by a relatively stable and simple data model (the 'conceptual schema')

(3) Implementation of various stored data structure techniques

(4) Explicit support for virtual information, validity/consistency checking and security checking

(5) Support concurrent use of database

In order to achieve these features and at the same time realize pipelining and functional abstraction in the system, the functional hierarchy is given the propsoed architecture as shown in Fig 1.5.

2.2 Data Models

From the discussion above, it is clear that selection of data models at the external, conceptual and internal levels has great impact on design of the interfaces between levels and functions to be supported at each level. This section presents an overview of research in the area of data models and points out approaches to be taken by the functional hierarchy.

2.2.1 Conceptual Data Models

2.2.1.1 Literature Overview

Recent research in the area of logical data models has followed

two directions. One has focused on enriching the conventional data models (e.g. hierarchical, relational, and network). It is argued that these conventional models are basically 'syntactic' data models which suffer from deficiency in semantic constructs. Numerous efforts have been made to enrich them, especially in the refinement of relational data model. The concept of 'normalization' in a relational model is an attempt to understand better the meaning of a relation by recognizing functional dependencies among the data <Codd72>. Schmid <Schmid75> further classified relations by 'type'. He suggests that, by indicating which type (e.g. entity type, association type, characteristic type) a normalized relation belongs to, the meaning of storage operations (e.g. insert, delete and update) on the relation are clarified. This concept of 'type' of relations enrich semantics of the strictly syntactic structure, and has motivated work on further normalization <Fagin77a>. The Entity-Relationship model proposed by Chen <Chen76> is also concerned with an improved modelling technique to be applied to real world facts. Smith and Smith <Smith77a & 77b> then added the concepts of aggregation, generalization and cluster membership attributes. Along the similar line, Bachman <Bachman77>, in an attempt to extend the network model, introduced the role concept in representation of real world entities. A comprehensive discussion of data model semantic constructs is found in the development of the Semantic Data Model (SDM) by McLeod <Mcleod78>, where additional concepts such as cover aggregation and event-type entities are included. A recent paper by codd <Codd79> has summarized these extensions in semantic constructs into a relational model called RM/T.

One way to summarize these developments is to plot them on a

one-dimensional chart (Fig 2.4), where at the left end there are strictly syntactic data models such as conventional hierarchical and network models, and more semantics are added to the data models as they progress towards the right, then at the right end of the chart there will be models proposed in the field of artificial intelligence, such as the Semantic Networks <Roussop75>, where an effort is made to provide the user with a powerful set of tools to model real world information as naturally as possible. Data models developed in the area of artificial intelligence also strive to provide flexibilities in naming a certain set of objects, depending on the context of the application and the angle from which information is viewed.

Another direction of research in logical data models emphasizes the identification of a basic simple construct. This construct, sometimes termed "minimum information unit", is simple, with clear and clean semantics, and may be easily collected in a meaningful fashion to represent complex varieties in semantic structures. Hierarchical and network models are considered too complicated for this purpose. They are not flexible in restructuring themselves to a different view. N-ary relations (or single-leveled files) and binary relations are more appropriate for use as the basic construct in this sense. However, the conventional n-ary relation is plagued by semantic ambiguity <Schmid75>. All the fields in a tuple are equally associated, while in the real world, some associations may be direct and others indirect. Placing all of them in a single tuple may lead to misunderstanding of the meaning of the information. Even though considerable efforts have been put in the concept of 'normalized' relations, it is felt that the best guard against spurious information is a binary association model.

It has been argued that the binary association or some close approximation has much more desirable technical properties than n-ary relations for use at the logical level <Senko77>. The advantages of a binary association data model are discussed in <Bracchi76, Falkenberg76>. Briefly, it is believed that binary associations have clean semantics, and are most flexible in supporting various external representations.

2.2.1.2 The INFOPLEX approach

It is not our purpose here to add to the debate of various data models. However we look into research in this area, and propose the use of a binary network type of model as the basis for logical design. The important qualities of a binary network are clean semantics and its ease in handling multiple-view support and mapping of the internal representations. In the remaining part of this section, a description of the proposed binary network model is given, followed by some examples demonstrating these qualites. We also believe that the binary construct is capable of supporting more complicated constructs demanded by some recent data models.

The Binary Network Model:

A visual presentation of our binary network (BN) model is shown in Fig 2.5. There are four basic constructs. Primitive Elements represent some objects or facts in the real world (Fig 2.5a). A Primitive Set is a group of primitive elements that have similar generic properties and therefore are given a common group name, called

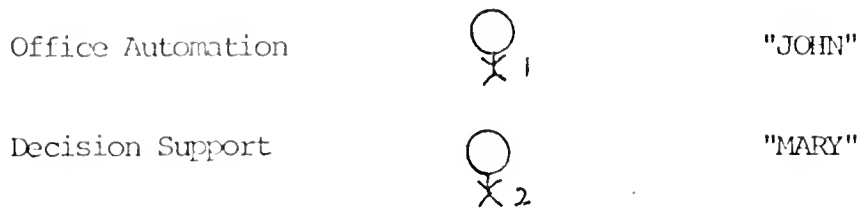


Fig 2.5a: Primitive elements

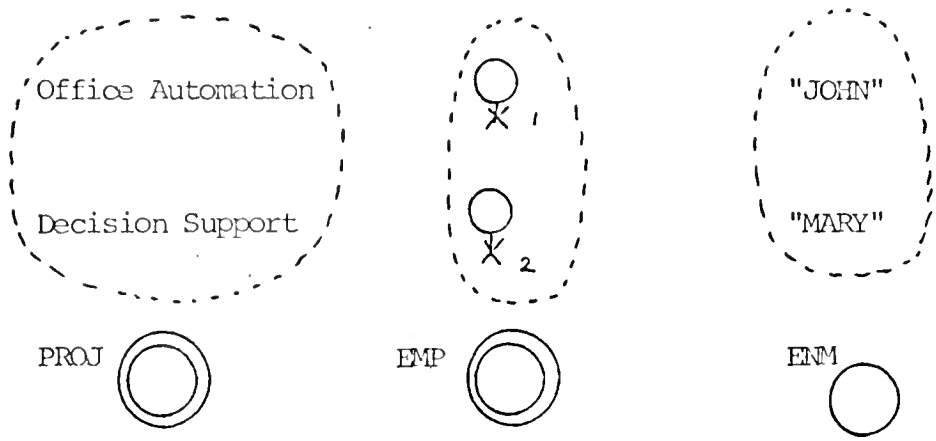


Fig 2.5b: Primitive sets (P_set)

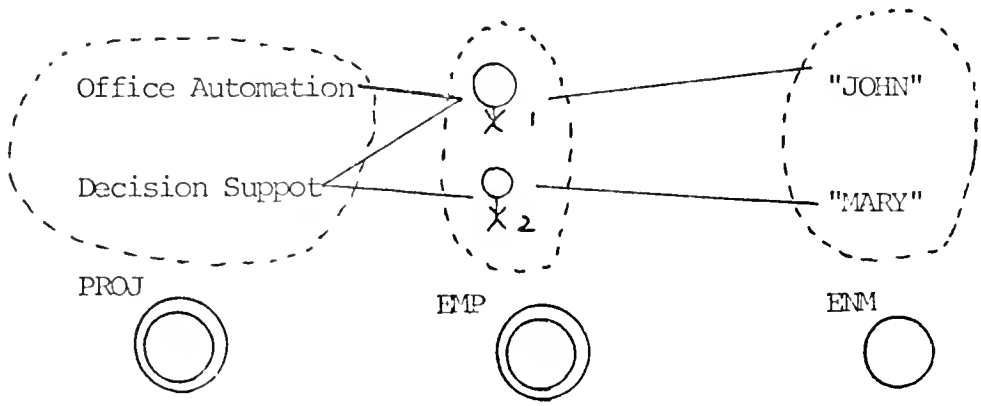


Fig 2.5c: Binary associations

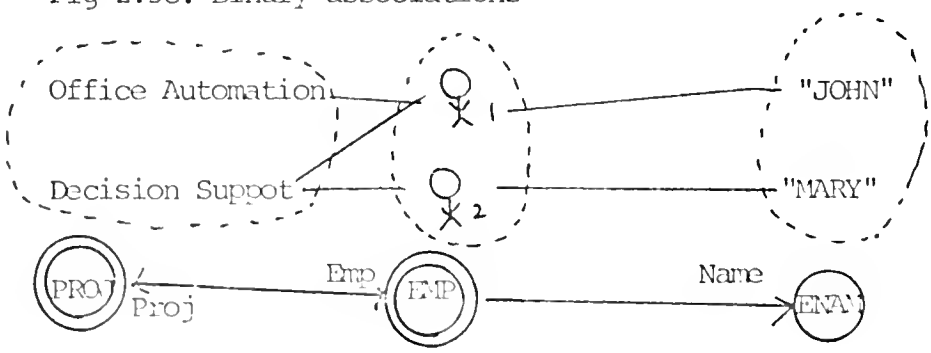


Fig 2.5d: Binary association sets (B_set)

a Primitive Set Name, or Pset_name (Fig 2.5b). Binary associations are representations of some real world relationships among primitive elements from different primitive sets (Fig 2.5c). A binary set is a group of binary associations that have similar generic properties (i.e., the incident primitive elements belong to the same primitive sets, and the associations have the same meaning). It is designated by a pair of primitive set names and a pair of association names (Fig 2.5d).

In Fig 2.5d, the upper portion (primitive elements and binary associations) represents the instance of the database, while the lower portion represents the schema of the database. Therefore, the schema of our BN model is composed of primitive sets (also known as the 'nodes') and binary association sets (also known as the 'arcs').

Further classification of nodes and arcs: In a binary network schema graph, a node can be either an entity node or a value node. An entity node serves to tie all equally related value nodes or entity nodes together. By 'equally related' we mean that those nodes tied to this entity node are all direct attributes of the entity node, instead of 'derived' attributes. An entity node corresponds to any set of real world objects (tangible or intangible) that have some common set of attributes which are revealed by the node's binary connections to other value nodes or entity nodes. It's own identity is reflected by these associations; i.e., an instance of an entity node does not have any value or identity, and its designation is made through instances of its associated nodes. In a sense, the purpose of an entity node is to collect equally related binary associations to form a semantically

clean n-ary association. Therefore an entity node is also refered to as an n-ary entity node. Those nodes that are not enity nodes are value nodes. Value nodes, in contrast to entity nodes, have values assigned to their instances.

Arcs can further be specified by several parameters. One is the syntactic function, which is given in terms of 1:1, 1:n, n:1 and n:m. The other, to be specified for each direction of the arc, is the semantic function, which is given in terms of 'hierarchical' or 'association', 'total' or 'optional', 'candidate_key' or 'non-key', etc. These parameters help further define and clarify the meaning of the storage operations on these nodes or arcs in our conceptual schema. For example, the instance of an incident node of a hierarchical arc depends on the existence of the associated instance at the other end of the arc for existence, while the association arc does not imply this restriction. A binary network schema with these distinctions is presented in Fig 2.5e.

Conditional Arcs: At the instance level, there are situations where the existence of an association depends on the value of the instance of another node. For example, an entity node PERSON may have an association with a value node TYPE, and if the value of the associated TYPE of an instance of the node PERSON is 'doctor', then this instance will have an association with another entity node DOCTOR; on the other hand, if it is associated with a TYPE 'nurse', it will have an association with another entity node NURSE. This means that the existence of the instance of an arc may depend on the value of another node. We shall distinguish this kind of conditional arcs from

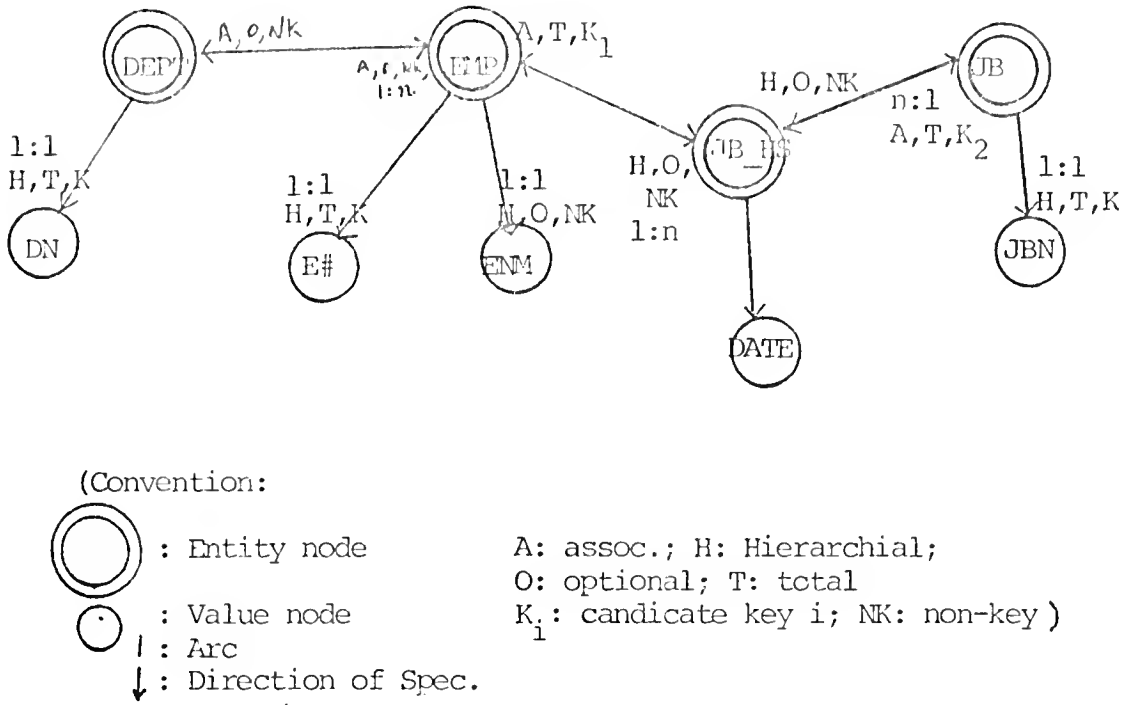


Fig 2.5e: An example of BN schema with distinctions of nodes and arcs

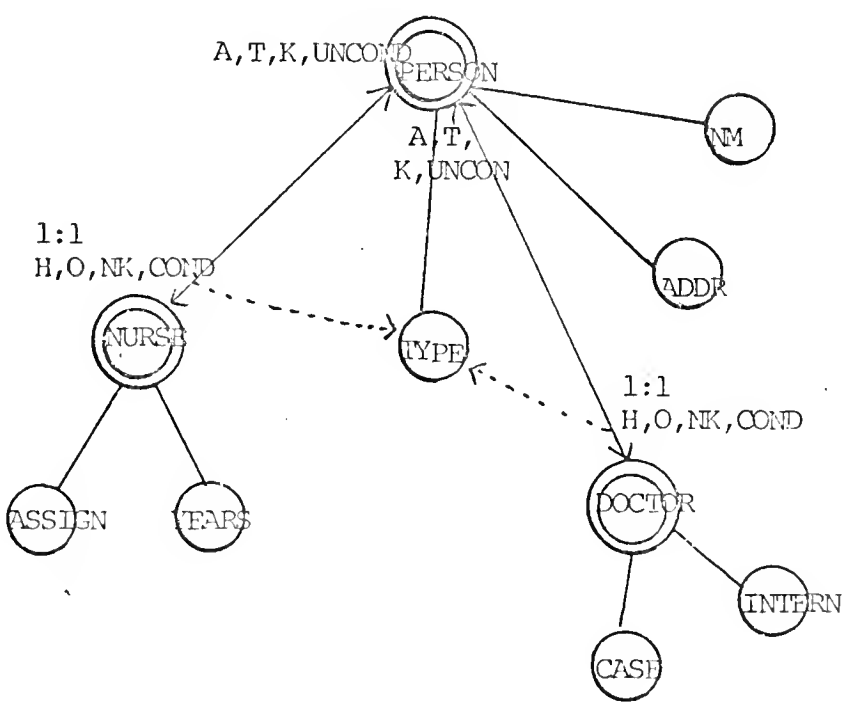


Fig 2.5f: An example of a BN schema with conditional arcs

the unconditional arcs. Fig 2.5f shows a diagram incorporating this distinction.

We shall conclude the description of the BN model by referring to its definition language specifications. Fig 2.6 is a BNF expression of the specification. Specifically, 'Define_Vset' will create a value node; 'Define_Nset' will create an entity node; and the attribute set in the Nset definition is manifested by creating arcs connecting this Nset to the nodes that correspond to the domain of the attributes. An example of a schema definition is given in Fig 2.7a, and its corresponding node-arc diagram is given in Fig 2.7b. It is believed that this definition language is very simple to understand and easy to use.

Implementation: The BN model is implemented at the N-ary level (the lowest level in the hierarchy that supports 'database semantics'). This level accepts the conceptual schema definitions in the form as shown in Fig 2.6, and generates the corresponding binary network. It keeps a catalogue of all the value, binary and entity sets defined in the schema and interpretes operations against the instances of these constructs. (More details are given in chapter 5, which describes implementation of the n-ary level.) The binary network is also made known to the internal schema designer as a basis for the file and access path design. The latter is to be specified in an internal schema specification language implemented at the internal levels. The binary network is also made known to the external schema designer to describe different types of views, which are implemented at the external view levels.

```
⟨BN statement⟩ ::= ⟨Vset⟩ | ⟨Nset⟩
⟨Vset⟩         ::= Define_Vset (Vset_name)
⟨Nset⟩         ::= Define_Nset (Nset_name, ⟨Attr_list⟩ )
⟨Attr_list⟩    ::= ⟨Attr_descrip⟩ | ⟨Attr_list⟩⟨Attr_descrip⟩
⟨Attr_descrip⟩ ::= (attr_name, ⟨domain_name⟩ , ⟨Arc_spec⟩ )
⟨domain_name⟩  ::= Nset_name | Vset_name
⟨Arc_spec⟩     ::= Syn , Sem , [⟨equivalence⟩], [ op_name ]
⟨Syn⟩          ::= 1:1 | 1:n | n:1 | n:m
⟨Sem⟩          ::= ⟨Hier⟩,⟨Imparative⟩,⟨Key⟩,⟨condition⟩
⟨Hier⟩         ::= Hier | Assoc
⟨Imparative⟩   ::= Total | Optional
⟨Key⟩          ::= Cand_key | Non_key
⟨condition⟩    ::= ⟨con⟩| uncon
⟨con⟩          ::= (attr_name, ⟨link⟩)
⟨link⟩         ::= (literal, Nset_name) |⟨link⟩,(literal,Nset_name)
⟨equivalence⟩  ::= eq= Bset_name | eq=Bset_name.R[2]
⟨Bset_name⟩    ::= Nset_name.Attr_name | Nset_name.op_name
```

Note: 1: [ ] denotes optional parameters; terminal symbols are underlined.
2: Bset_name.R refers to the reverse of Bset_name

Fig 2.6: A BNF[1] specification of the schema definition language

(a)

```
Define_Nset (EMP,
             (E#,E#,1:1,H,T,K,uncon)
             (DEPT,DEPT,n:1,A,T,NK,uncon))

Define_Nset (DEPT
             (DN,DN,1:1,H,T,K,uncon)
              EMP,EMP,1:n,A,O,NK,uncon,eq=EMP.DEPT.R))

Define_Vset (DN)
Define_Vset (EMP#)
```

(b)

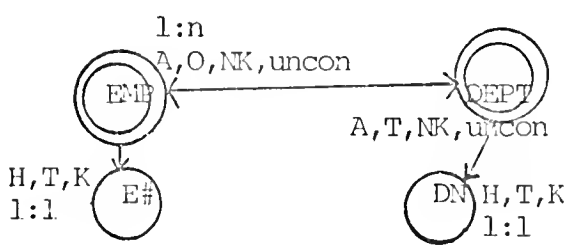


Fig 2.7a & 2.7b: An example schema definition and it BN graph

Examples: Some examples are given here to demonstrate the advantages of the Binary Network data model:

(1) Clean Semantics: As illustrated in Fig 2.8, an n-ary relational schema may potentially carry ambiguous information, while a binary form of the schema eliminates this ambiguity.

(2) Ease of mapping into different constructs: It is awkward to map an n-ary relational schema containing one-to-many relationships into its equivalent hierarchical form (Fig 2.9a), while the mapping is performed more naturally from the Binary Network schema (Fig 2.9b). Also, since all the binary connections are explicit, it is easier to maintain certain semantic constraints. (For example, deleting a product will trigger deleting of the shipments of that product.)

(3) Ease of mapping into internal constructs: Fig 2.10 shows how the binary network may be mapped into a variety of internal data structures by simply specifying how nodes and arcs are to be implemeted; while this convenience does not exist in most other types of schema representation.

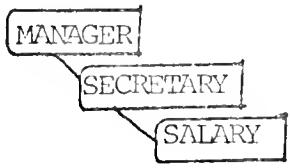
Support of rich semantics: This subsection summarizes the BN model's capability of supporting rich semantics. This is done to show that the BN model, while parsimonious in its constructs, can be used as the basic building block for richer models.

(1) multi-valued attributes: This is achieved simply by specifying the syntactic function of the arc implementing this attribute to be 1:n. No other explicit specifications such as 'characteristic entities' are necessary.

An ambiguous n-ary schema:

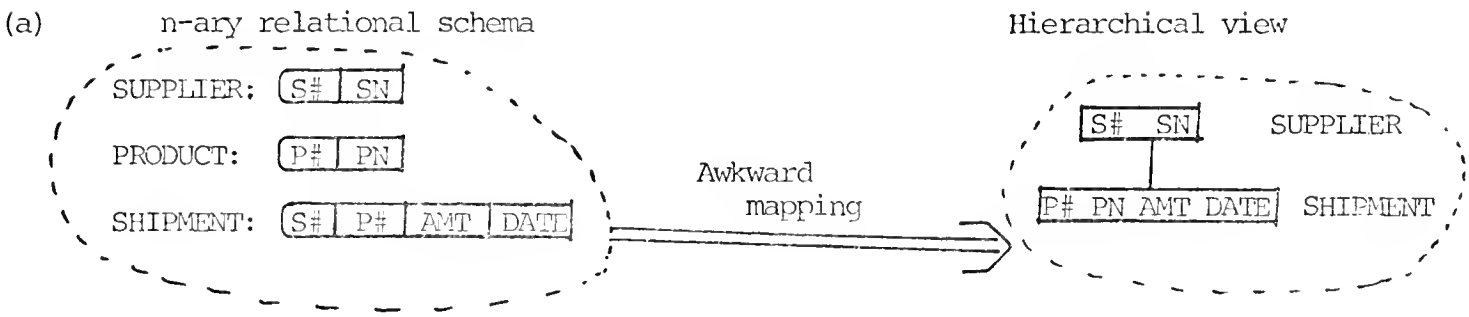
(Does the SALARY refer to the MANAGER's SALARY or the SECRETARY's SALARY?)

Elimination of ambituity through the use of the binary form:



(SALARY refers to the SECRETARY's SALARY)

Fig 2.8: Binary network safeguards clean semantics



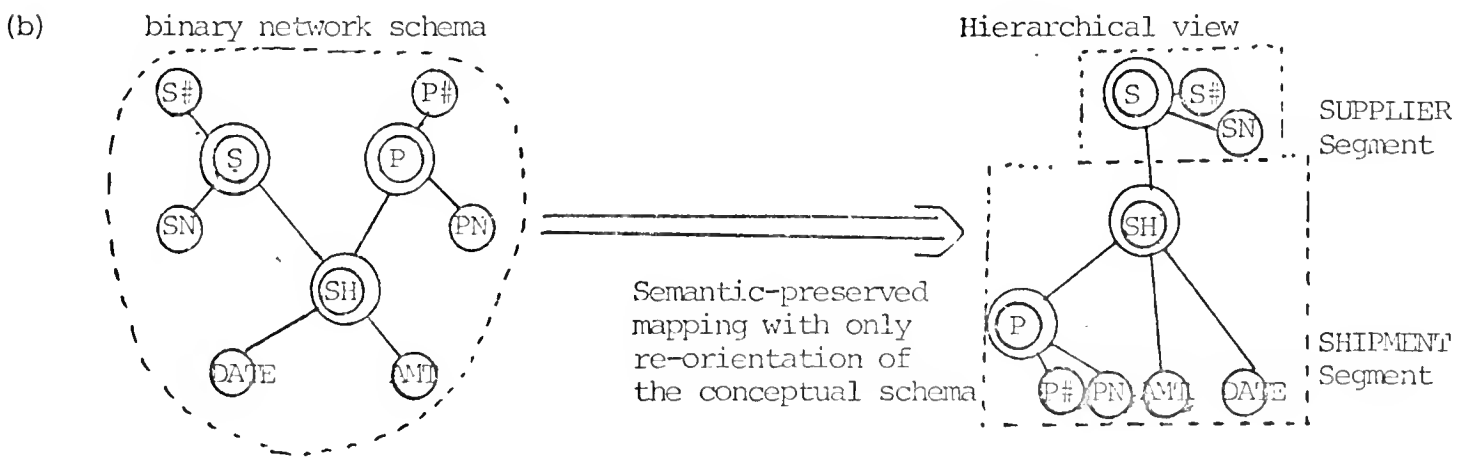


Fig 2.9 (a) & (b): Binary network facilitates mapping of views

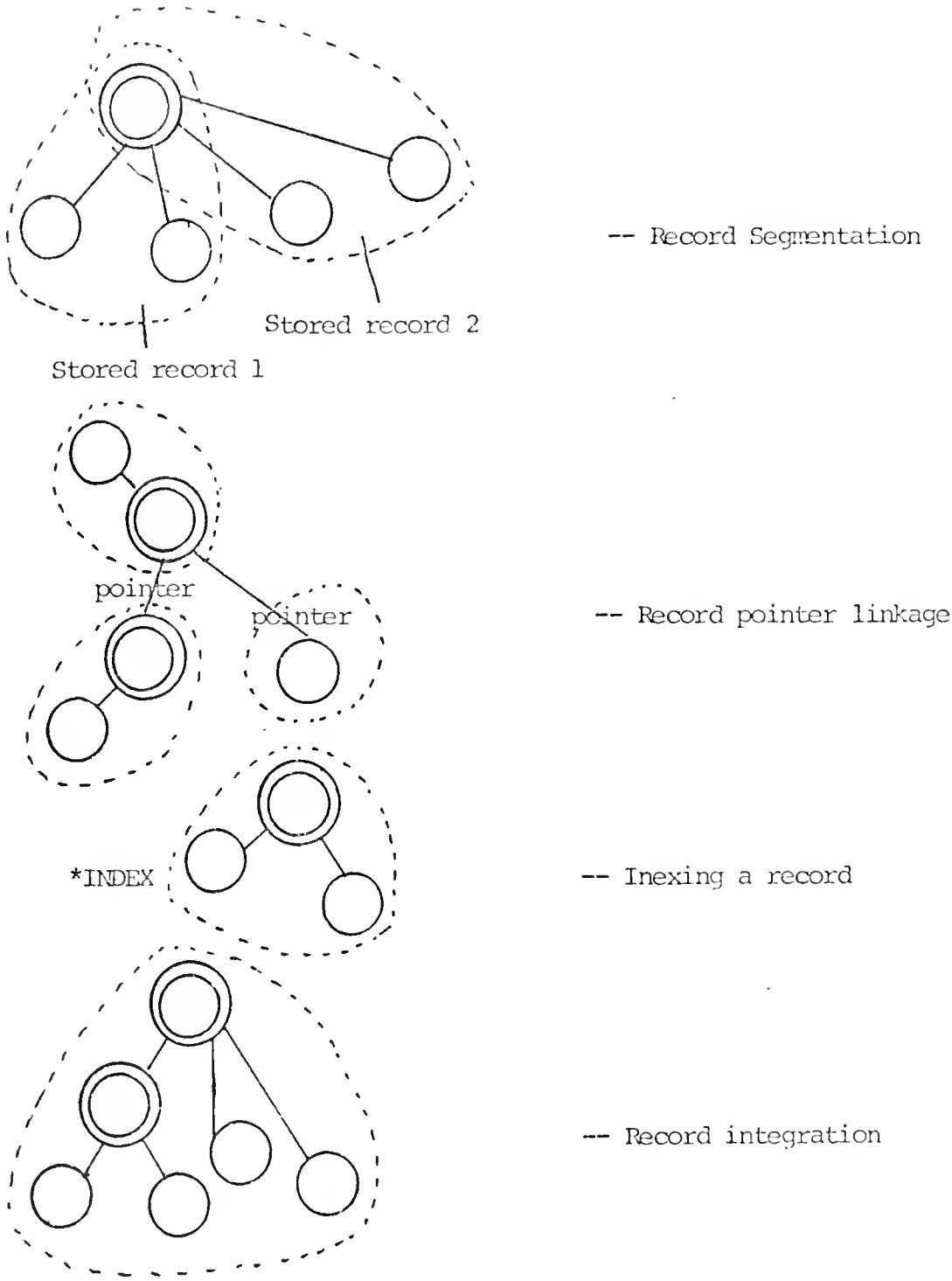


Fig 2.10: Examples of internal specification using binary network data model

(2) aggregation: This is achieved by specifying the domain of the aggregated attribute to be another entity. (There is no restriction which says that the domain of an attribute has to be atomic).

(3) generalization: This is achieved by the conditional arc property. In the case of a an unconditional generalization hierarchy <Smith77a>, the association from a node at a higher level to that at a lower level is based on a conditional arc, while the association from lower to higher is based on an unconditional arc. In the case of an alternative generalizaiton <Codd79>, the associations in both directions may be conditional. This scheme also implements cluster membership attributes and the role concept.

(4) hierarchical association: This is achieved by specifying the arc to be 'hierarchical'. It clarifies the fact that this association provides external identify to the 'child' node, and deletion of an instance of the 'parent' node necessitates deletion of all the associated instances of the 'child' node.

Summary: Our approach bears certain resemblance to concepts of 'atomic semantics' and 'molecular semantics' introduced in <Codd79>. In that paper, simple n-ary relations are referred to as atomic semantics, while molecular semantics represent 'bonds' that tie up atomic semantics to form complex constructs. In a similar spirit, we propose to use binary associations as 'atomic' semantics. We also move ahead to show how these atomic semantics are actually implemented at the internal levels, and how they are collected to realize more complex logical structures at the semantic construct levels. Moreover, we will

sent the formation of different views for the end user from the underlying binary structures.

2.2.2 The Internal Data Model

2.2.2.1 Literature overview

The internal data model is used to describe physical data structures of a database. The choice of a physical data structure is the outcome of a physical database design process, which uses the conceptual schema and statistics on usage of the database to generate either an optimized or a 'good' physical data structure. The goal of physical database deesign is good performance, i.e., good throughput and response time, under a certain access/update pattern and load on the database.

The scope of physical database design spans the file structuring problem (e.g., sequential file or inverted list), the access path selection problem (e.g., sequential scan or indexing), the record segmentation and allocation problem (e.g., the number of fields in a physical record), and the reorganization problem. The problems of memory hierarchies and allocating files among storage devices are sometimes included in the physical database design, but they are not addressed at the internal levels of the Functional Hierarchy.

There has been a great deal of research in the area of physical database design. This results in a desisre to support a large number

of data structures in a database management system. A general survey of these structures is given in <Date77>, and a survey of physical database design methodologies is given in <Schkolnick78>.

In order to support a large number of data structures, the internal data model has to be very general, i.e., it has to be a model through which the various data structures may be described by the user and implemented by the DBMS. While most of the research in data models has been dedicated to conceptual data models (as indicated in the previous section), some prominent ideas have been generated in the context of the data translation and conversion problems <Smith71> and in the development of the DIAM system <Senko73>.

In the DIAM system, the concept of a basic encoding unit (BEU) is introduced. A basic encoding unit is a unit of data stored in the computer. It is comprised of control information and a data field. The former may be further broken down into the identifier field, the attribute field (i.e. length and encoding type) and the relationship pointer field (Fig 2.11). The idea is that, by manipulating definitions of the control information parameters of a BEU, various data structures can be realized. This provides a powerful media for describing data structures, and a common basis for implementing them. The implementation will consist of functions that decode the parameters and build up data structures accordingly.

The concept of BEU summarized the attempts up to then to generalize all data structures in a single construct (i.e., an encoding unit), and allowed variations to be parameterized. Use of the BEU

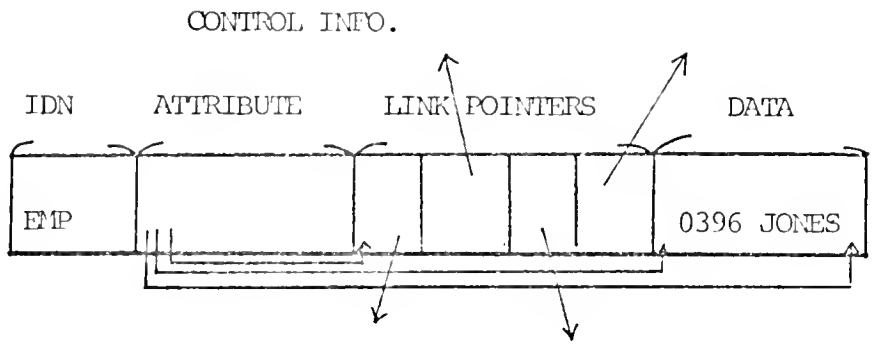


Fig 2.11: Format of a BEU

concept is extended and further formalized in a paper by <Fry77>. The authors of that paper adopted this concept to express the "translator view" in their data tranlation project conducted at the University of Michigan. They call it the Logical Encoding Unit (LEU). Several operations are defined on the basic construct:

1. Collapsing/expanding: this pair of operations encode and decode data values into bit strings;
2. Extracting (factoring) / dispersing (distribution) : the first operation condenses the encoding unit by bringing common fields into a catalogue entry. It also may specify how relationship pointers are expressed (i.e. whether by actural pointers or by physical contiguity, etc.). The second operation does the reverse.

2.2.2.2 The INFOPLEX approach

We have adopted the BEU approach to internal modelling because of its power and simplicity. It is considered fairly general in its ability to encode various data structures, and at the same time very neat to work with. In chapter four, we will describe how the BEU model is used to describe and implement the physical data structures of a database formatted in terms of the BN conceptual data model.

2.2.3 Multiple-View Support

As discussed in section 2.1, a sophisticated DBMS ought to be able

to support different external views of the database. This is important on two grounds:

(1) Protection of investment in existing application programs: Most of the existing application programs are either written in a conventional environment without a database system or implemented on a database system that employs a different data model (e.g., IMS). It is important that a DBMS is capable of 'simulating' the old data structures so that the exisiting application programs do not have to be rewritten from scratch. This practical consideration is critical in implementing conversion from one DBMS to another.

(2) Diversity in views in different applications and by diferent users: Each user's view of the real world may differ depending on the application context and the preference of the individuals. In <Nijssen76> it is pointed out that selection of an application data model by the user is analoguous to selection of a religion. Therefore an effective DBMS should be capable of providing the 'freedom of choice' by supporting diversity of views.

Supporting multiple views requires: (1) view modelling and view integration during the logical database design, and (2) specification and implementation of the mappings between the external views and the conceptual view. The first one has been discussed in the context of the logical database design process <e.g., Chen76, Bernstein76, Navathe78, Vetter77>. In fact, it is very related to the development of a concpetual data model which must be used to describe the 'integrated view' of the database as a result of view integration. The

second one, on the other hand, is an issue in the design of the database management system, and is to be incorporated into the external view levels of the functional hierarchy. In essence, the external view levels are responsible for accepting definitions of the views in terms of different data models (e.g. relational or hierarchical, etc.) and their structural and operational mappings to the conceptual schema based on the binary network model, and translating operators issued against the external data models to the equivalent conceptual schema operators. These are problems to be addressed in this section.

Scope of the problem:

In order to clarify the mapping problem, three levels of complexity of the multiple-view support are defined here:

(1) Subschema: This is the simplest level of the mapping problem. A subschema is a view that represents strictly a subset of the conceptual schema. For example, if a relational model is used in the conceptual schema, the allowable external views are also relational, and each individual view contains relations that are subsets (in terms of either degree or cardinality) of those defined for the conceptual schema. The subschema facility is extremely useful for security control, and does provide certain degree of data independence. But it does not provide views expressed in different data models to fully accomplish the objectives described in the beginning of this section. Examples of this kind of facility are DBTG's Sub-schema facility <DBTG76>, IMS's logical database facility <IMSa>, and System R's view

facility <Astranhan76>.

(2) Simulating a different external data model: This level of mapping actually involves more than one data models. For example, in the research of the Database Computer (DBC), it has been shown that the DBC data model can be used to accomodate relational, hierarchical or network type of external models by incorporating explicitly the idiosyncratic information about these external models into the DBC record-oriented data model <Hsiao79b>. Another example is the implementation of a non-relational data model on top of System R by incorporating a sequence-number field into the relations <Astrahan76>. This level of multiple-view support has generally ignored the possible interactions between different external models due to the explicit altering made to the conceptual schema. (This is largely due to the fact that the conceptual schema in question is syntactic-oriented rather than semantic-oriented.) It is one step above the subschema approach, but may still not be ideal in supporting multiple types of models simultaneously.

(3) Transformation: This is the most ambitious level of the multiple view support, and is the kind that the functional hierarchy strives to achieve. It supports multiple types of external data models simultaneously. The basic premise of this kind of view support is that the conceptual schema is an embodiment of all the knowledge available, and the external models are merely different templates for abstraction and transformation of this knowledge. As pointed out in <Falkenberg77>, the process

of realizing the external views is analoguous to the 'de-conceptualization' process discussed in linguistics. Also in <Klug77>, the correspondence between the external and conceptual levels is described to be similar in nature to the implementation of one abstract data type in terms of another.

The implications of implementing the last type of multiple-view support are the following:

(1) Three levels of abstraction for the mapping process are to be considered <Rothnie76>. At the highest level there is the data model correspondence, which specifies the mapping language for the two different models. At the data definition level (i.e., the schema level), an association between the exteranl schema and the conceptual schema is defined in terms of the mapping language. At the instance level, the mapping must perform the translation of the external operators into the conceptual operators.

(2) The mapping has to cover both structural correspondence (basically name mapping) and operational correspondence (i.e., translation of operators).

(3) There has to be control over interference among different views and proof of correctness of the mapping.

Correctness of Mapping: There has been relatively little discussion in the literature about correctness of mapping. In <Paolini77>, a need for formal definition of mappings is proposed. It also gives an example of interactions among external models. Borkin <Borkin78> furthered this study by providing a set of formal

definitions for data model equivalence.

INFOPLEX approach: Support of multiple external data models in INFOPLEX results in multiple external view levels in the functional hierarhcy. These levels are basically parallel to each other, rather than organized hierarchically. Each level is designed to provide all the views expressed in a particular external data model. Presently, three data models are supported: the relational, the hierarchical, and the network models.

In chapter six we shall show how the structural as well as operational mapping between the three types of external models and our conceptual schema are specified. Proof of correctness will be conducted as one of the future research dimensions. Basically, to show that a mapping is correct, the following is to be demonstrated:

> Given a conceptual data model C, an external data model E, a set of conceptual operators C_{op} and a set of external operators E_{op} we would like to construct a strucutral mapping language M and an operational mapping algorithm M_{op} such that

$$E = M(c)$$
$$E_{op}\{E\} = M_{op}(E_{op})\{M(c)\}$$

2.3 Summary

We have discussed the formation of the general structure of the functional hierarhcy in the context of recent research in the database management systems. We have examined the literature from both architectural and functional points of view and identified functions and their organization in the functional hierarchy. Research in the area of logical data models and physical data models is reviewed to shed light on the structures to be supported by the levels of the functional hierarhcy. Finally, the need for multiple-view support and problems associated with it are discussed.

III. MEMORY MANAGEMENT

As described in the previous chapters, the INFOPLEX database computer consists of two hierarchical components: the storage hierarchy, which is a collection of storage devices implementing a large virtual storage, and the functional hierarchy implementing database management functions. There is a virtual storage interface in between these two components. We now start at the lowest level of the functional hierarchy -- a level that interacts with the storage hierarchy through the virtual storage interface and manages the large virtual storage address space. We have isolated memory management as a separate level because it deals with physical memory issues (e.g. bytes and byte addresses) which are very different from logical issues (e.g. logical units of data) of a DBMS. This level is depicted in Fig 3.1.

3.1 The id approach

The primary task of the memory manager is to manage a vast volume of virtual storage while insulating the rest of the system from the details of virtual memory management. An approach using the data id is proposed here. We first give a brief description of this approach and then provide some rationales.

This method divides the entire memory into pages. When a piece of data is stored in a page, it is given an id. The id comprises the page number and a pointer slot number within that page. The idea is an extention of the TID <tuple id> concept used in System R (Astrahan76).

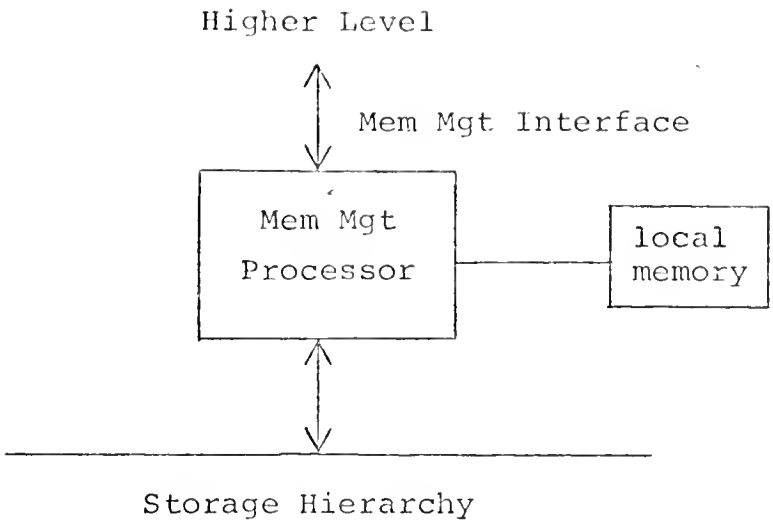


Fig 3.1: Memory Management Level

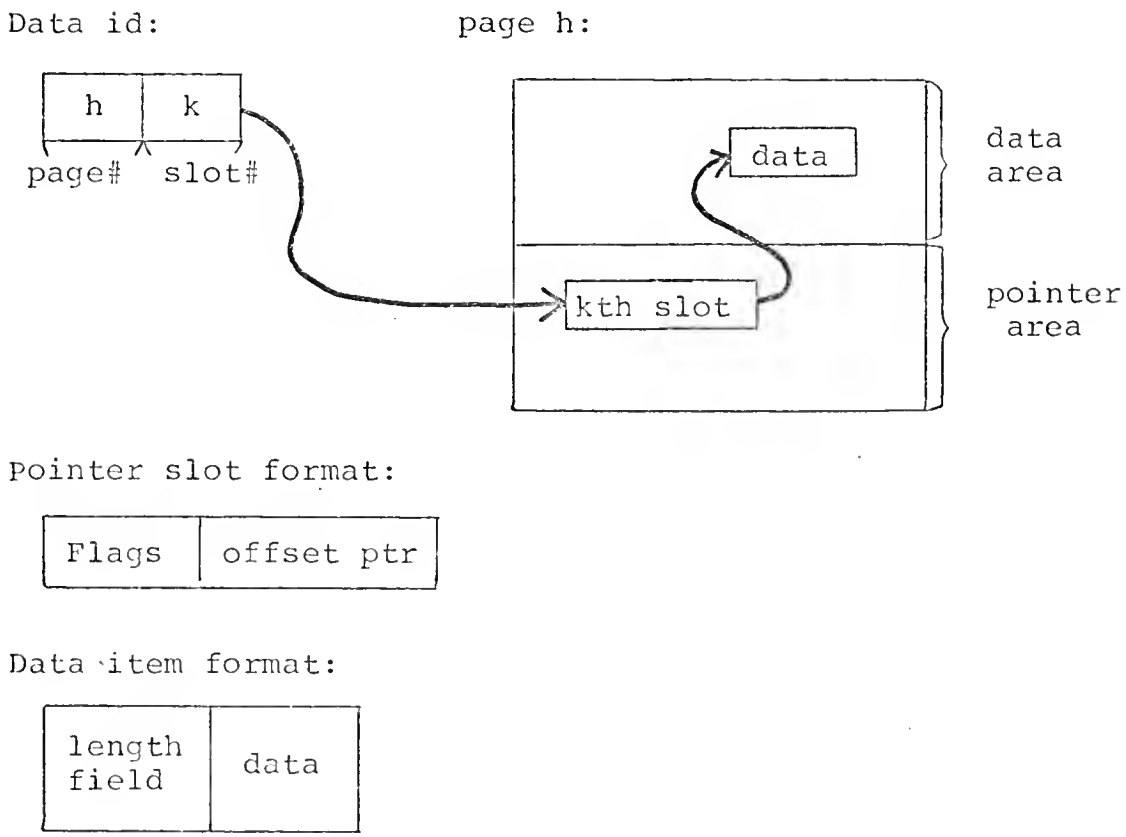


Fig 3.2: The id scheme

The pointer slots reside in a special region, called the pointer area, within each page. Each pointer slot contains an offset within this page. The concept is presented in Fig 3.2.

If a data item is moved around within the same page, only the offset needs to be updated, and the id remains unchanged. If a data item is moved to another page, a flag is set in the original slot, and the pointer slot actually points to the data item's new page/slot. This operation is illustrated in Fig 3.3.

If the data element is to be moved to a new page for a second time, its original pointer slot (h,k) is updated and (1,m) is released (see Fig 3.3). In order to support this process, the original id (h,k) must be stored along with the data element each time it is moved to a new page.

Using this approach, the memory manager isolates the virtual storage from higher levels of the system, i.e. higher level modules do not have direct access to the virtual storage. Levels above the memory manager reference data solely via the id, and are not concerned with physical 'byte addresses'. Advantages of this approach are:

(a) The length of the id may be much smaller than that of the byte address of the virtual memory.

(b) It provides a separation between the logical identifier of the data (i.e. id) and the physical identifier (i.e. the byte address) such that when data are moved around in storage, their id's need not be affected.

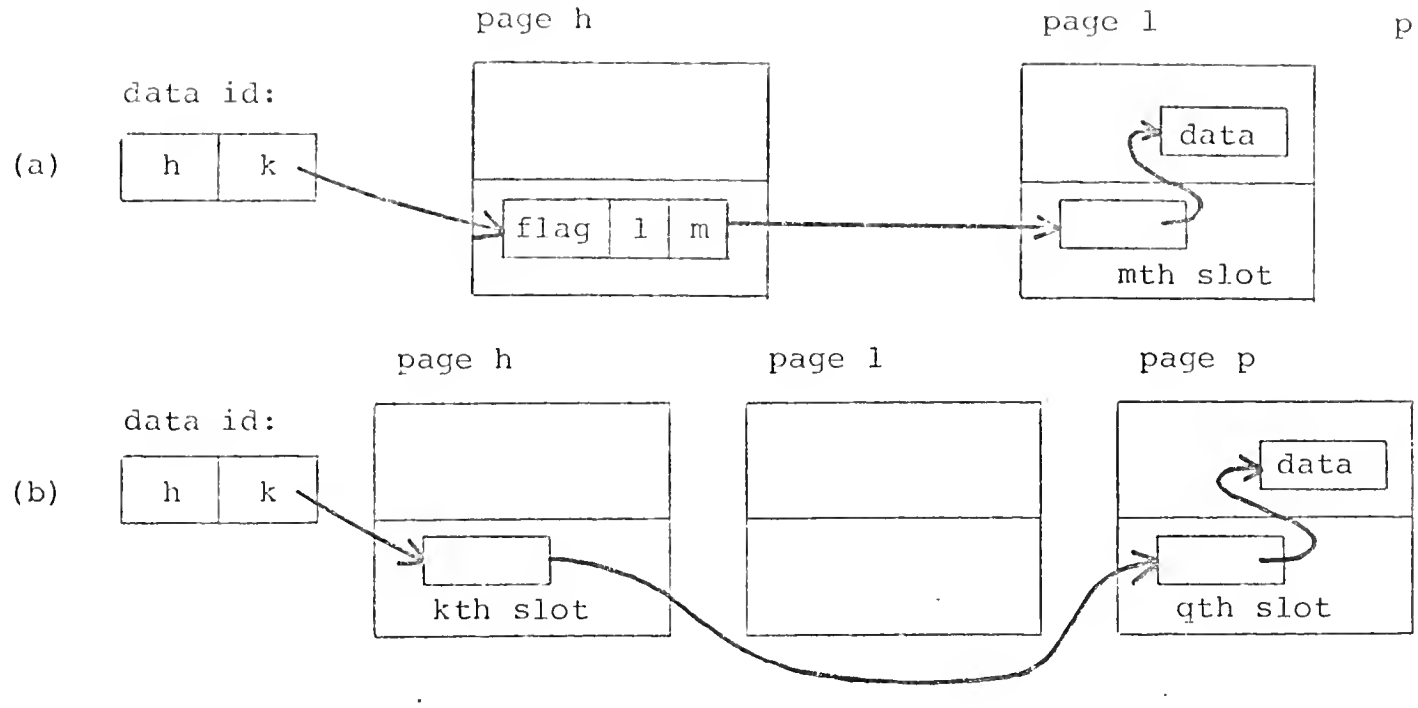
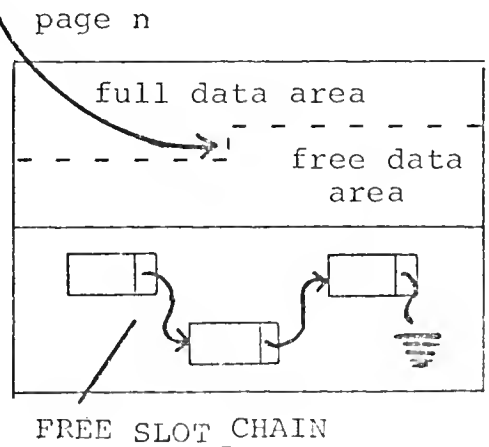


Fig 3.3: Data item moved across pages with id unaffected

page catalogue

PAGE#	RESERVE	F_SLOT	F_DATA	F_SPACE	D_COUNT
0					
1					
⋮					
n		k			



RESERVE: Page reserved for special purpose

F_SPACE: Size of free_data_area

D_COUNT: Size of deleted area

Fig 3.4: Keeping track of available storage space

(c) The memory manager insulates the 'byte detail' from other levels, centralizes the memory management algorithms and policies, and therefore reduces the complexity of other levels and eliminates contention involved in shared usage of the storage hierarchy.

3.2 Allocating storage space

When a request to store a data item is received, the memory manager has to (1) allocate some free storage to this data item and (2) store it and update pointers. In order to keep track of free storage space, the memory manager has to have a catalogue. One method is to keep a table, in which each page has an entry, which points to the first free slot in that page. All other free slots of the same page are chained together. The catalogue also contains an offset of that page which indicates the starting byte of the free data area. This is shown in Fig 3.4.

Data items are of variable lengths. Therefore, a length field is stored with each item. When an id is presented to the memory manager for retrieval of the item, the length of the item is first examined and the number of bytes to be pulled out determined. The length field itself may be of variable length to accomodate a wide range of sizes of data items.

For simplicity, a data item is usually not split across page boundary. When the size of the free data area of the first available

page is not enough to store a particular data item, the next page is approached, until a page that is capable of holding this data item is found. Another field may be added to the page catalogue, which indicates the size of the free data area of each page.

3.3 Page reservation and clustering consideration

Even though it is definitely advantageous to have storage allocation decisions centrally made at the memory management level, there are situations in which, for practical reasons, certain areas of the storage space are requested to be set aside by higher levels. Once pages are assigned to be dedicated to certain purposes, they can no longer be used for storage of items outside of these purposes. For example, some pages may be reserved to store only elements of a set which are stored according to a hashing function. The higher level actually calculates the id of a data item to be stored and passes the id to the memory manager, instead of having the latter assign the id. Special parameters are incorporated into these commands.

3.4 Virtual Storage interface

It is through this interface that the memory manager interacts with the Storage Hierarchy. The latter provides a byte-addressable memory, and STORE/LOAD operations are performed as within a conventional computer. Details of virtual storage operations are completely concealed beneath this interface and are responsibilities of

the storage hierarchy. The memory manager simply views itself as equipped with a memory of a very large size.

3.5 Memory management interface

The memory manager provides the following functions for modules of higher levels that call for service (refer to Fig 3.5 for their logic):

operations	arguments
CREATE	(mode, byte_string, id0)
UPDATE	(id, new_string)
DELETE	(id)
RETRIEVE	(id)
RESERVE	(code, no_of_pages)

CREATE is invoked when a data item (i.e. a byte string) is passed to be stored in the database. Other parameters concerning the data item's storage area may be passed at the same time. There are 3 modes for CREATE: (1) In Regular mode, the caller does not care where the item is to be stored.

(2) In id mode, the caller specifies the id of the item to be created.

(3) In approx mode, the caller provides the id of another item around which this new item is to be created.

UPDATE replaces the old content of the data element designated via id

with the new byte string passed. If the new string is of a smaller or the same size of the old one, it is written over the old one. However, if the new byte string is larger, the area where the old string is stored is discarded, and a new free data area (preferably in the same page) is sought to store it. In either case, the id is not affected.

DELETE is effected simply by chaining the pointer slot to the free slot chain and setting a flag in the slot. The data area freed by DELETE is not recaptured until a page compaction module is invoked to walk through pages to collect them. In order to facilitate page compaction, the number of bytes deleted in a particular page is recorded. When this counter exceeds a critical value, a flag is set for this page, and a request for compaction is filed (see Fig. 3.4).

In addition to functions that may be invoked through the interface, there are miscellaneous housekeeping tasks to be maintained. Page compaction is one, and statistics collection and data reorganization may be another. Other design issues such as page size, data area size, sizes of pointer slots and length fields, as well as whether an overflow area is to be reserved for the page, etc., are to be discussed in detail design.

One final consideration at this level is the place where the page catalogue is to be stored. Since the storage hierarchy is directly accessable at this level, it seems natural to use part of this virtual storage to store the page catalogue. An adequage number of some pre-determined pages may be assigned to the page catalogue, and entry of the catalogue is retrieved by the following formula:

base address of catalogue + (page no - 1) * size of entry

The structure and information to be stored with the catalogue are to be determined during the detail design. In general, it opens a question as to how large catalogues are to be maintained in functional decomposition, since the set of catalogues represents a very large database, and data structure manipulation functions devised to maintain the database may also be needed to housekeep catalogues. The page catalogue represents a design problem and different alternatives and their tradeoffs are to be explored.

Fig 3.5: operational commands at memory management level

```
(note:  Arguments suffixed by '*' are return arguments)
LOCATE (id, byte_addr*, code*)
     1.  calculate slot addr of id;
     2.  load content of slot;
     3.  if free flag set, then code = 'free', and return; else
     4.  if new page not set, then go to 7; else
     5.  use info in current slot to obtain new slot address;
     6.  load content of new slot;
     7.  calculate addr of data item and store it in byte_addr
     8.  return (byte_addr, code='o.k.')

CREATE (mode, byte_string, id0, id*, code*)
     id mode:
     1.  LOCATE (id0, A, R)
     2.  if R not equal to 'free', then Return (Code='contention');
       else
     3.  id = id0;
     4.  F_CHAIN('remove , id, l=length(byte_string));
     5.  LOCATE (id0, A);
     6.  store byte_string at A;
     7.  Return (id0, code='o.k.');
     approx mode:
     8.  (assume id0=(p0,k0))
          p=p0; l=length (byte_string);
     9.  if Reserve (p) not set, and F_space(p) >= l,
         then go to 11; else
     10. p=p+1; go to 9;
     11. let k=F_slot(p) and id = (p,k);
     12. go to 4;
     regular mode:
     13. let p=PAGE;
     14. if F_space(p) >= l then go to 11; else
     15. p=p+1; go to 14;

              (Note:  PAGE is a variable that points to an
                        immediately available page)

DELETE (id)
     1.  LOCATE (id, A, Code);
     2.  let L1=length of data element at A;
     3.  F_CHAIN( 'insert',id,L1);
     4.  if new page flag not set, then return; else
     5.  let id1=id of new slot;
     6.  F_CHAIN('insert',id1);
     7.  return;

UPDATE (id, byte_string)
     1.  LOCATE (id, A, Code)
     2.  let L1=length of data element at A;
     3.  if L1>= length (byte_string) then go to 8; else
     4.  let id0=id and call CREATE ('approx', id0, byte_string, id);
     5.  if id and id0 are of the same page, then update content of
         slot designated by id0; and call F_CHAIN ('insert', id0, L1),
```

```
        and go to 9;
        else
     6. set new page flag at slot designated by id0;
     7. go to 9;
     8. store byte_string at A;
     9. return;

F_CHAIN(op, id, L)  where op='insert' or 'remove'
    assume id =(p,k).
     1. if remove op, then remove kth slot from free slot chain.
        update F_data(p) by adding L to it, and if p full, declare it;
     2. if insert op, then insert kth slot into free slot chain of
        page p;  increment delete_byte_counter by L.  If critical
        value exceeded, add page p to compaction request queue;  set
        free flag of that slot.
```

IV. INTERNAL STRUCTURE

4.1 Introduction

In chapter two, we have discussed the choice of the BEU concept for internal modelling. We shall, in the present chapter, expand this concept and show how the internal levels of the functional hierarchy are designed to support the binary-network conceptual data model with various data structure techniques.

Our approach produces a gradual mapping of the internal construct to the conceptual data model. The highest level of our internal structure, the binary association level, may be viewed as the lowest level of the conceptual model itself. The data definition language to be accepcted by this internal structure level is simply the binary definition loosely coupled with parameters that guide internal construct building. These parameters are checked for consistency and then distributed to levels that are of concern. For example, parameters specifying how many indexes are to be maintained for a particular set of elements are processed by the unary set processor, while those specifying how data is to be edited before being stored are processed by the internal encoding level. It is easy to show that changes in techniques of internal representation can be accomplished by changes in the parameter space presented to the internal schema writer. The parameter space is virtually the collection of tools available to the database designer. While these parameters may change, the conceptual definition remains stable. This parameterization approach is an example of how a true separation of the conceptual schema and the

internal schema may be brought about.

In our design, the following specifications are made to the BEU's:

1. A BEU is the smallest logical unit of data to be stored and retrieved. They are grouped into sets, called Unary Sets A unary set is a collection of generically similar BEUs. By 'generically similar' we mean that they share common control information which has been factored into the catalogue entry of the unary set. The identifier field (which is used to name the unary set) is replaced by a link to a catalogue entry (i.e. the identifier field is 'factored'). This link may be a pointer, a table, or via physical contiguity. It is to be specified by the internal schema writer.
2. Binary relations are implemented by association links. The meaning of these links are also factored into the catalogue entries. Binary links may again take the form of actual pointers, physical contiguity or data duplication.
3. We break the relationship pointer field of the BEU into two areas, one called SP area (Set Pointer area) and the other AP area (Associative Pointer area). It is obvious that an encoding unit has to exist before its associations to others may be created. Therefore we follow a natural route that breaks the task of managing these two types of connection into two hierarchical levels. One is called the unary set processor level, and the other, built on top of the former, the binary association level.
4. The stored representation of the data value field of the encoding unit may be very different from that of the unit being processed at various levels of the system. This specification, if

common to encoding units of a certain set, may be factored into a catalogue entry of that set. We identify the task of stored representation transformation as a very different task from the relationship management. Therefore a level called data encoder is isolated for this job.

To conclude, the format of our BEU takes the shape as shown in Fig 4.1.

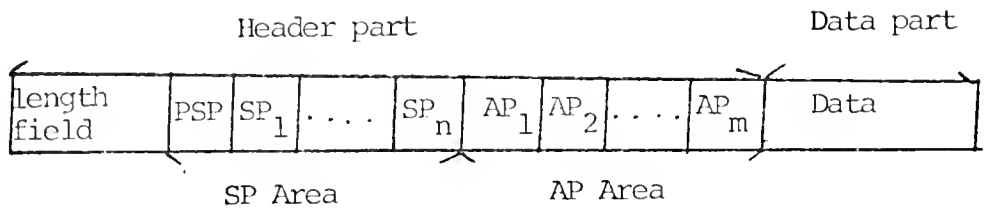


Fig 4.1 Format of a Basic Encoding Unit (BEU)

4.2 Data Encoding Level

In this section, the data encoding level, and the next one, the unary set level, the phrase 'set', unless otherwise qualified, refers to the unary set, while the phrase 'element' or 'data element' refers to a BEU.

The data encoding level is singled out to implement various techniques in data encoding and text editing, such as suppressing of blanks and duplicated characters in the text, other text compaction techniques, crptographic methods to encode data for protection, etc.

An element to be stored is passed to this level along with the set it belongs to. A catalogue is traversed to determine whether it is a set of which the data part is to be encoded according to some specified function. If it is not, the element is stored as it is; if it is, the encoding function is located and transformation performed on the data part of the element (refer to Fig 4.1) before it is stored. A flag of a stored element is set if it has gone through encoding, and a reversed procedure (i.e. decoding) is followed when this element is retrieved.

4.2.1 Data Definition Interface

A set that requires data field encoding will have an element of the Encoding Structure Parameter Space (ESPS) coupled in its definition, as shown in the following:

Define_set (Setname, other parameters, u€ESPS).

This parameter u is then given to the data encoding level to build a catalogue, with the set name serving as the key entry. Various types of data encoding methods may be precoded into this level, each given a name, and may be invoked by giving this name. Data encoding is then accomplished by executing the procedure that implements the method.

To facilitate fast retrieval of catalogue entries, a set name may be hashed to generate the address of its catalogue entry. If the catalogue is small, it may be stored in the working memory of this level; if it is large, the virtual storage may have to be used to accomodate it. In either case, an entry is made up of the set name and encoding method name. The latter is represented by a pointer to a procedure to be excuted. Procedures, again, may be stored either in the working memory or the virtual storage.

4.2.2 Operational Interface

Requests to create, delete, update and retrieve an element are passed down from higher levels. An element is distinctively composed of a header part, which is intact at this level, and a data part. Also passed as an argument is the set name of the element.

In short, this level sits between the unary set processor and the memory manager to perform transformation of the data field of an element. Clearly, the system will still function without this level.

It represents an option presented to the user. When this level exists, the encoding methods that it supports may also differ from one system to another, depending on the needs of the user.

4.3 Unary Set Level

4.3.1 Introduction

The purpose of this level is to link elements into sets and facilitate fast retrieval of an element in a set.

The meaning of the "set" may need to be clarified first. Every data element stored in the storage hierarchy belongs to one and only one Primary Set. The set processor maintains a catalogue of all sets defined. These sets may be logical unary sets defined by the user or sets defined by modules at higher levels to store information for housekeeping purposes. Therefore, "set", to the unary set processor, is merely some collection of data elements that share certain common properties. Every stored element in the database is uniquely identified by the combination of a set name and the content of the element.

Every catalogue entry serves as the 'head' of a primary set. An entry contains information concerning implementation of a set. It contains a pointer pointing to the first instance of its member, and other information, such as sort, index, hashing and physical contiguity, used to implement retrieval mechanisms. The format of a data element when passed to this level is shown in Fig 4.2.

Together with the element, the set name to which this element belongs is also given to the set processor. Accordingly, the set processor pulls out the catalogue entry of this set, concatenates a set

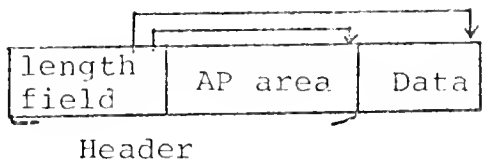


Fig 4.2: Format of an element as it is passed through unary set interface

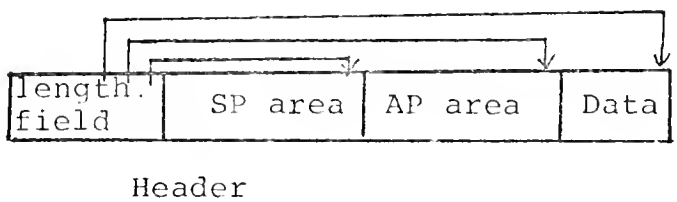


Fig 4.3a: Format of an element after "SP" area is added to it

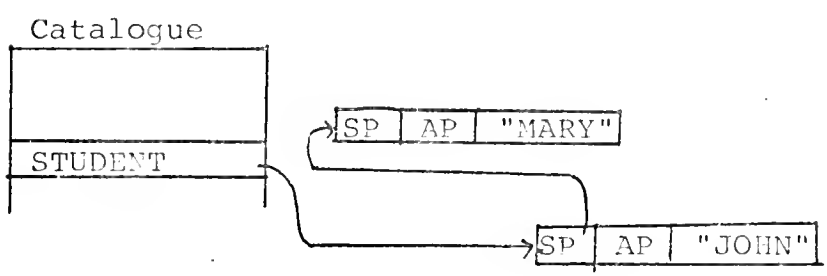


Fig 4.3b: Inserting an element into a primary set

chain pointer (SP) field to the element, and inserts this element into a proper position in the set. This is shown in Fig 4.3a and 4.3b for sets that are implemented as linked lists.

4.3.2 Primary Sets and Secondary Sets (Subsets)

There are two types of sets implemented at this level. One is the primary set, chained by the primary set pointer (PSP). A member of a primary set is created by actually storing a data element into the data base.

The other type is the secondary set. Inserting an element into a secondary set is by way of passing the id of the element (i.e., the element is already stored), and the secondary set it belongs to. There is a catalogue entry for each secondary set defining the structure of the linkage of this set. A set of this type can be considered a subset, in contrast to the primary set discussed above. This mode of set processing is very useful in implementing binary associations of the form 1:n or n:1. It makes the retrieval mechanism implemented at this level available to subsets of elements as well. An example is given in Fig 4.4.

4.3.3 Catalogue implementation

Catalogue entries by themselves are members of a primary set by the name of CATALOGUE. Techniques used to implement sets and

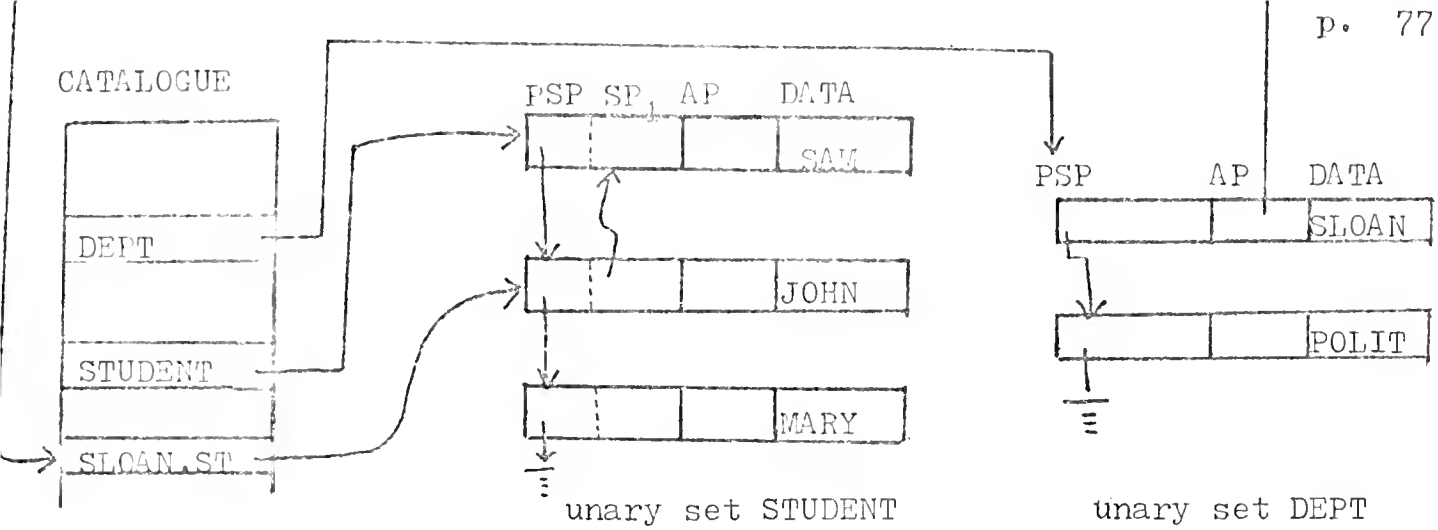


Fig 4.4: Subsets - Binary association between unary sets DEPT and STUDENT is of the type 1:n. In this example, while SAM and JOHN have AP's pointing to SLOAN, SLOAN's AP points to a catalogue entry SLOAN.ST which chains SAM and JOHN together with a SP_1.

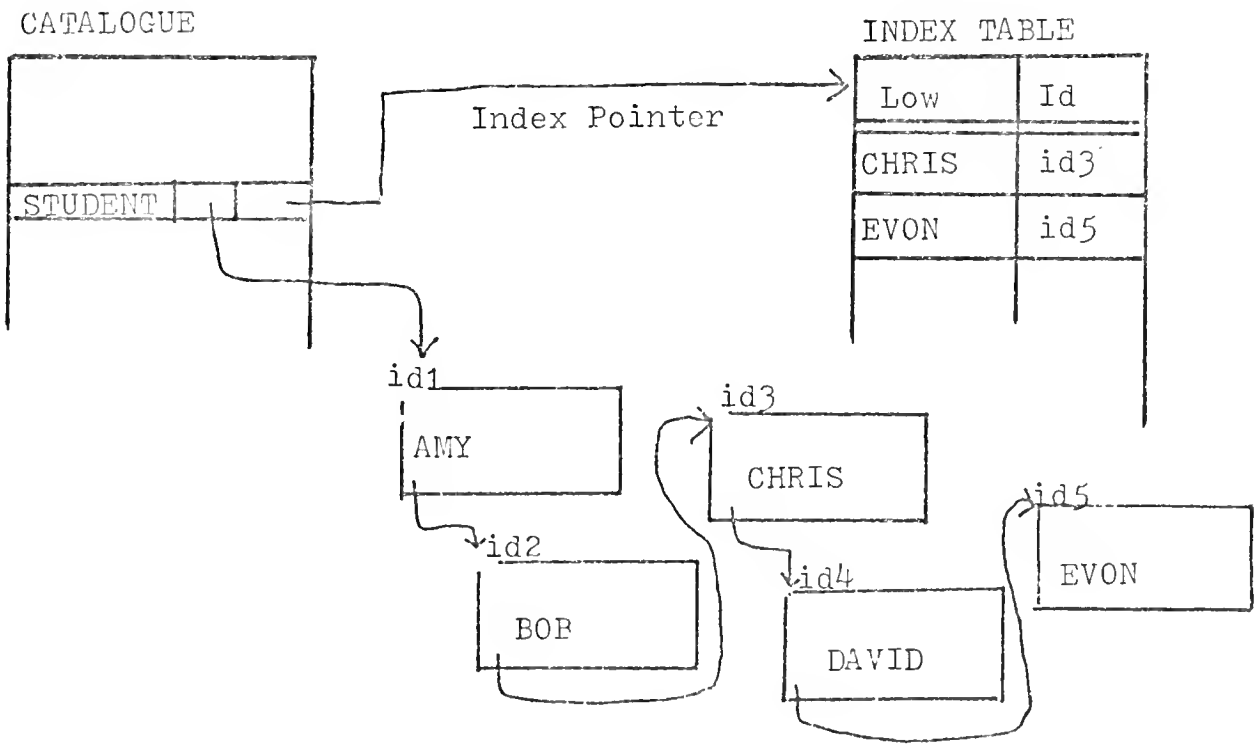


Fig 4.5a: Sorted linked list and its index table

facilities available for search and retrieval at this level can be employed to process the catalogue as well. To illustrate, the first Data Definition command to the set processor, DEFINE-CATALOGUE, is a statement which defines the structure of the catalogue set. Most likely, for example, the catalogue set is hased. After the structure of the catalogue set is defined, a command to define a regular unary set is transformed into an insertion command which inserts the catalogue entry of this set into the catalogue set; likewise, when a catalogue entry is to be retrieved, a retrieval command is used to accomplish this job.

4.3.4 Fast Search Mechanisms

The set processor is responsible for presenting a stored element to a caller, given its set name and data part. It may also be required to accomplish sequential retrieval of a particular set. The internal schema, therefore, specifies how a set is to be implemented in order to accomplish this retrieval task; for example, whether a sorted linked list is desired, whether it is a two-way or one-way link, whether an index is to be built on the data part (or part of the data part) of the element, and whether a scatter table is to be maintained for the hashed data part. If an index is requested, the set members are usually sorted, and when a member is inserted or deleted, index entries, if affected, are updated. Other parameters may be added to determine the structure of the index table. A pointer to the beginning of the index table is maintained in the catalogue entry of that set. If a scatter table is specified, the hashing function as well as the beginning of

the scatter table are stored with the set catalogue entry. Other techniques may be incorporated by augmenting the parameter space of the catalogue entry.

4.3.4.1 Sorting

Sorting is used to facilitate sequential processing and indexing. A module SORT is used to perform this task. To make this mechanism more powerful, sorting can be based on the entire data part or part of the data part. The sort field may or may not be unique. It may even be desirable that sorting be performed according to the data part of elements of another set whose id's are part of the data part of the set to be sorted. These different modes of sorting are specified when sets are defined, and indices built accordingly.

The sort module is invoked after the database is first loaded. Then the sorted set is maintained by logic incorporated into INSERT, REMOVE and UPDATE functions. It may be invoked during the operational time of the database to reorganize sets that are previously unsorted, or sorted with another key.

4.3.4.2 Index Table Implementation

Suppose we have a sorted linked list as shown in Fig 4.5a, and an index table on the right is built for this set. Index table may be implemented in several ways. If it is by physical adjacency, then the whole table may be considered as a sorted set implemented by physical contiguity and stored away. When search in the table is desired, the

entries of the table are retrieved the same way members of a set are retrieved, and decoded according to its structure parameters (e.g. the length of each table entry) that are stored with the index table. These parameters are passed when this set is defined by DEFINE-SET.

Another approach would be to build the index table as a sorted linked list, and then make use of functions designed to manipulate linked lists to manipulate entries of the table. A multilevel index may also be built. If the lower level index table is built as a set, then the higher level index table is merely an index on this set. An example of indexing by linked list is given in Fig 4.5b, and a multi-level indexing example is given in Fig 4.5c.

When removal of an item in an indexed set is requested and if that element is a member in the index table, the table has to be modifed. A module that builds index tables (called BUILD-INDEX) is periodically called to reconstruct the index table as the set is augmented. For example, a counter may be incremented when a delete or insert is done on a set, and the module BUILD_INDEX is called when this counter reaches a critical value. During the database load period, this mode of calling can be suppressed and indices built only after the database is fully loaded.

Essentially, BUILD-INDEX would visit every element of the set and select elements at a particular interval to be entries in the index table.

4.3.4.3 Hash Table Implementation

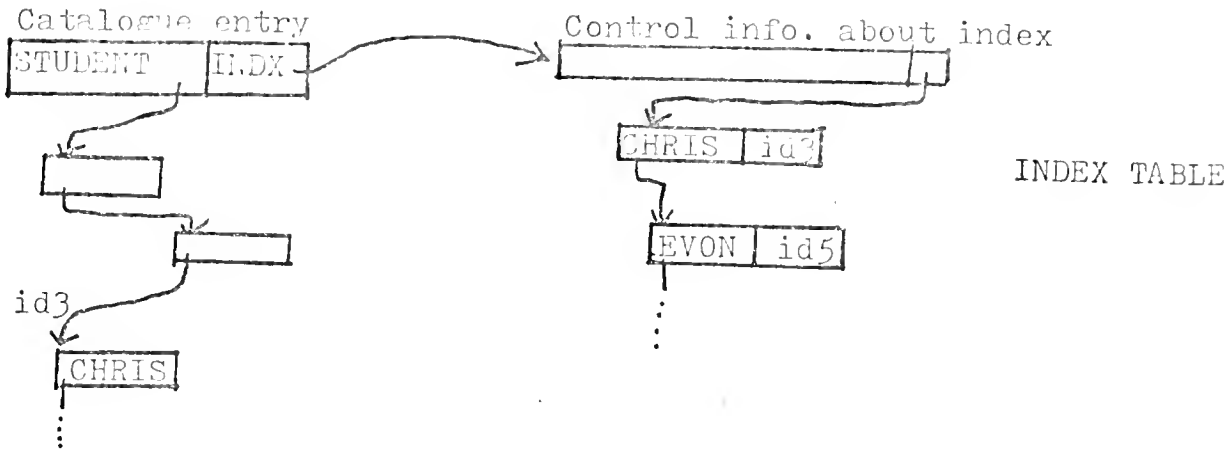


Fig 4.5b: Index table implemented as elements of a set

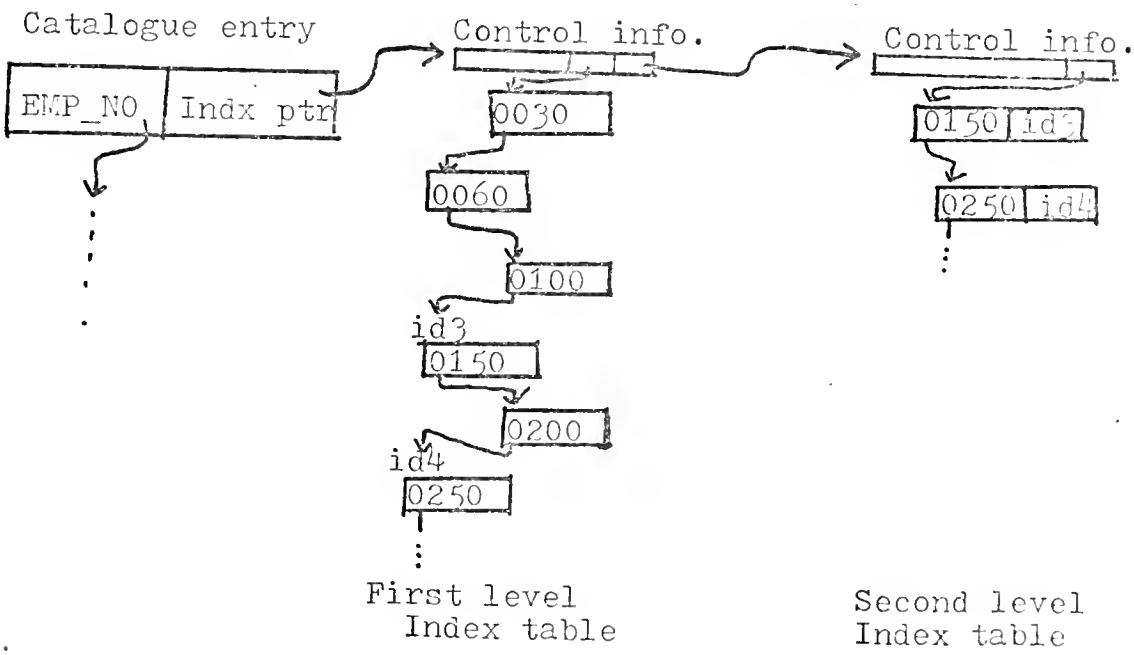


Fig 4.5c: Multi-level index table implementation

The hash pointer contained in the catalogue entry points to a location where the hashing function and other parameters are defined. One approach is to assign a certain series of pages that are to be used to store this particular set. The hashing function is performed on the whole or part of the data string, and returns a slot number and the lower positions of the page number as well. The essence is to generate an id number which belongs to the area assigned to this set. This area is reserved for this set only and no other sets are to be stored within it.

When collision occurs, either a pointer chain or a linear search starting from the collided id can be used to handle the problem.

Again many of these properties may be parametarized together with the internal schema of the set involved. One also has to be careful when the data part of an element in a hashed set is modified. For example, if JOHN is modified to be JOHNNY, and if

H(JOHN)=0100

H(JOHNNY)=0907,

then in the new location (0907) a flag is set to point back to the original. The logic of hashing is incorporated into INSERT and UPDATE.

4.3.4.4 Summary of Fast Search Mechanisms

The salient feature of this level is the fact that approximately all techniques for internal representation that are geared toward fast search or retrieval of an element of a set can be parameterized and incorporated into the structure of the set. What has been shown is an

example implementation, in which sets are primarily stored either as linked lists or by physical contiguity, and search mechanisms include linear search, indexed sequential and hashing. However, other facilities may be provided if the parameter space of the structure of the unary set is augmented.

4.3.5 Data Definition Interface

This is the interface across which sets as well as their types, retrieval mechanism and encoding format are defined. The structure of the SP area of an element is also given. The data definition language of unary sets is given in Fig 4.6; a typical definition of a unary set looks like the following:

DEFINE_USET (Uset_name, other parameters, $x \in SSPS$, $u \in ESPS$)

Where u is to be passed to the data encoding level, and x is processed and entered into the set catalogue at this level. Two other commands DELETE_USET and UPDATE_USET are used to modify set definitions. (SSPS stands for Set Structure Parameter Space).

4.3.6 Operational Interface

It is through this interface that operational manipulations of data elements are made. Commands are provided to insert, retrieve, delete and update members of sets. Arguments include the set name, id or data part, and/or header part. A list of commands at this level is to be found in Fig 4.7. To provide a feeling as to what operations may

be involved when these commands are invoked, some general logic is also given in Fig 4.7

4.3.7 Conclusion of Unary Set Level

A summary of modules identified at this level is given in Fig 4.8. Some modules are implemented on top of others, and inter-connections between modules are delineated according to the module logic outlined in Fig 4.6 and Fig 4.7.

Fig 4.6: DD interface at unary level

(1) Define_catalogue: This command defines the structure of the unary catalogue.

(2) Define_Uset (Uset_name, x ϵ SSPS, u ϵ ESPS): This command defines a unary set, where SSPS stands for Set Structure Parameter Space, which may include specifications of the following parameters:

a. set type: primary set or secondary set (i.e. subset);

b. set element storage location: this parameter specifies how the storage location of each element in this set is determined; there are 3 modes:
(1) id mode: the location of each element in this set is determined by higher levels;
(2) hashing mode: the location is to be determined by hashing the data part of the element;
(3) system mode: location determined by element link described below;

c. set element link: This parameter specifies how elements in a set are linked together; there are 3 different ways:
(1) pointer: by way of link list;
(2) pointer sequential: by way of sorted linked list;
(3) physical contiguity: by way of id contiguity;

d. index: this parameter specifies the number of indexes to be built. For each index thus required, the following information is furnished:
(1) Which part of the element is to be indexed? (It may either be part of the AP area or part of Data area);
(2) Full indexing or partial indexing? If partial indexing is used, how sparse is it going to be?
(3) Will any sort previously performed on this set be useful?
(4) structure of the index? (i.e. the location and entry size, etc.)

e. additional sort and indexing: a set may be sorted (by link list) based on different keys, and further indexes may be built. These are specifed in a similar way as described in d. above.

(3) Update_uset (Uset_name, xϵSSPS, uϵESPS); and

(4) Delete_Uset (Uset_name): These two commands effect changes or deletions of a catalogue entry. The former may force internal data re-organization, while the latter may involve deleting all the elements in the set. Ramifications will be studied and detailed.

Fig 4.7: operational commands at unary level

```
Create_element (Uset_name, (AP, data) or id0, id*):
     This command creates a unary element.

     1. retrieve catalogue entry by
         Retrieve_element ('data' mode, Uset_name, ctl_entry*);
     2. decode ctl_entry;
     3. case 'set element storage location' of
        id mode: id=id0;
        hashing mode: do;
           perform hashing on data;
           generate id;
           format SP area and then BEU;
           try: try to store this element at location  id by
                   CREATE ('id' mode, return_code*);
                  if return_code = 'contention', then
                  call Collision_Handler(id*) and go to try;
           end;
        system mode: continue;
     4. case 'set element link' of
        pointer or pointer_sequential: do;
           format SP and BEU;
           try to store this element by
           CREAT ('regular' mode, id*);
           end;
        physical_cont: do;
           obtain Last_id and Inc from Ctl_entry;
           if id out of bound of reserve area of this set, then
           call Reserve_more;
           id=Last_id + Inc;
           format SP and  BEU;
           store BEU at id by  CREAT ('id' mode);
           end;
     5.  if set element link is pointer then
         call Insert (Beg_pointer, id);
         else if set element link is pointer sequential then do;
            call Search (Uset_name, data, fnd*, id1*, id2*);
            call Insert (id1, id, id2);
            end;
     6.  if additional link list sorts and indexes are specified,
         then update the lists and indexes;
     7.  return (id);


Retrieve_element (mode, Uset_name, full_data or partial_data or id,
     (AP, datar, idr)*, code*);

     data mode:
     1. call Search (Uset_name, full_data or partial_data,
           fnd*, id1*, id2*);
     2. if fnd = null then return (code = 'not_found');
        else do;
```

```
        idr=fnd;
        call Retrieve_element ('id' mode, fnd, AP*, datar*);
        return (AP, datar, id);
        end;
 id mode:
1. RETRIEVE (id, byte_string*);
2. decompose byte_string into SP, AP and datar;
3. return (AP, datar, idr=id);
```

Search_element (Uset_name, patial_data or full_data, fnd*,idl*, id2*)

This subroutine locates elements in a set. It is also responsible for making an intelligent decision about which of the following access paths to use (if available):

1. hashing
2. indexed
3. indexed sequential
4. binary search
5. linear search

Information necessary for this decision making is stored in the unary set catalogue. (When only partial data is specifed, and if more than one elements contain that partial data, all of them will be located and returned, unless otherwise suppressed by the caller).

Delete_element (mode, Uset_name, full_data or partial_data or id,

code*)

This command removes an element or a group of elements from a set. If the Uset_name is a secondary set, only the connections related to the secondary set are removed. That is to say, only the part of the SP of the element that is used to chain this element to the subset is affected. If the Uset_name is a primary set, then this element is deleted from the database, so are its connections to all subsets. Special attention is paid to connections made through hashing or physical contiguity.

Update_element (id, new AP, new data)

This command replaces the old content of the element designated by id to the new content specified within the command.

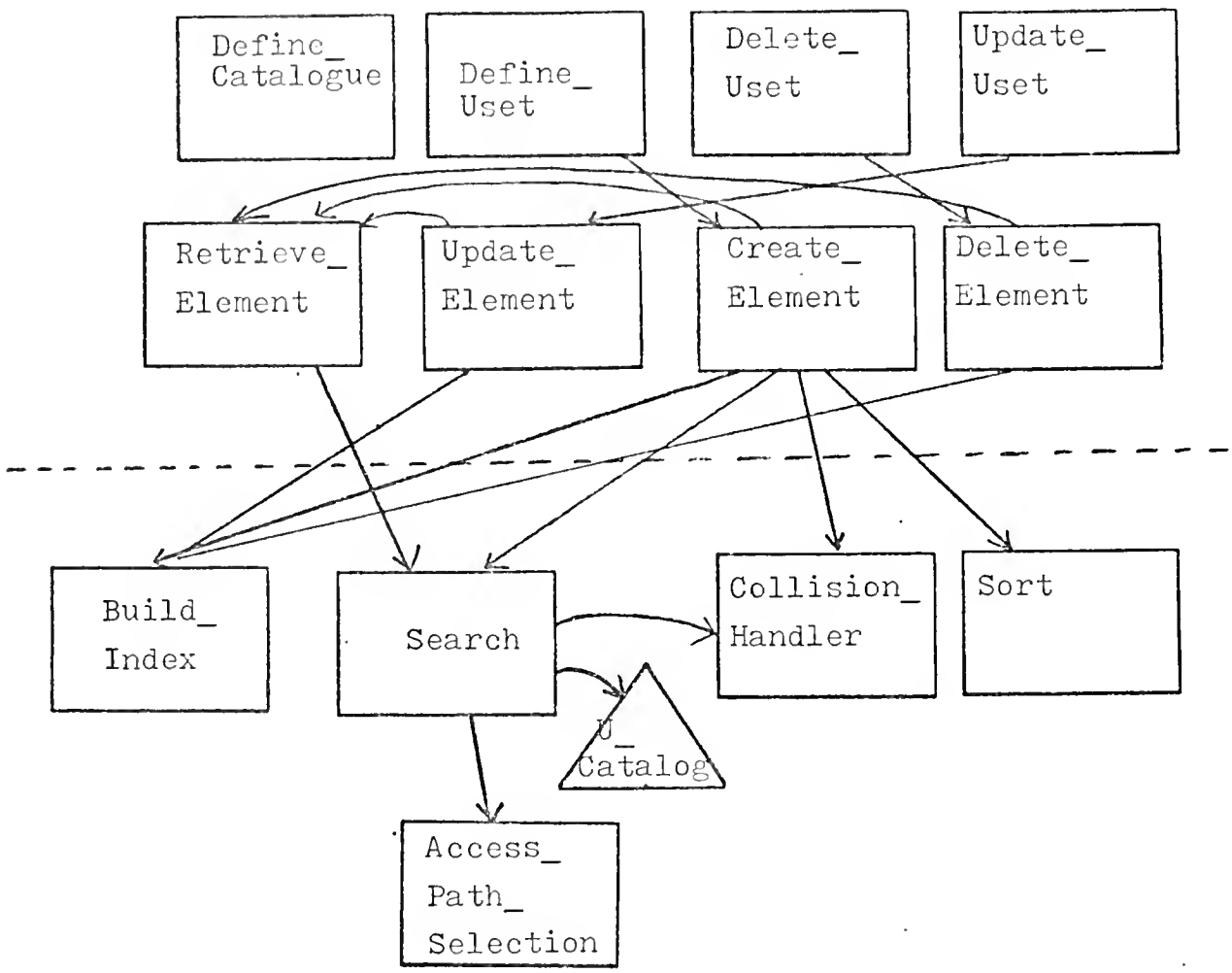


Fig 4.8: Unary level sub-modules

4.4 Binary Association Level

4.4.1 Introduction

This level implements binary associations specified in the conceptual schema. It serves as the bridge between the conceptual and the internal models. On the one hand, it communicates with higher levels in terms of 'information units', such as primitive sets and binary relations specified in the conceptual schema; on the other hand, it talks to the lower levels in terms of 'stored elements' such as BEUs and unary sets. The essence of this mapping is briefly summarized below:

a. A primitive set in the conceptual model is usually (but not necessarily) mapped to a unary set in the internal model. Therefore a primitive element is usually implemented by a BEU. However, there are situations in which a primitive set does not correspond exactly to a unary set. For example, as will be explained with details later, when a primitive set is to be embedded in an associated set, it will not be mapped to a unary set. Rather, its existence is manipulated through the unary set that implements the embedding primitive set.

b. Binary associations of a primitive element are implemented within the AP area of its BEU.

4.4.2 General Mechanism

The function of this level is to implement binary associations among primitive elements. It keeps two catalogues; one, called CTLP, describes the collection of primitive sets defined for the database, and the other, CTLB, contains information concerning binary relations among these sets. An entry of the latter is composed of names of the sets involved in the binary relation, their reciprocol attribute names, the function type (e.g., 1:n or n:m, etc.), and the association structure. Based on these structure specifications, unary sets and their formats are defined.

Recall that a stored element, a BEU, is composed of a Set-Pointer (SP) area, an Association-Pointer (AP) area, and a data part. The SP area is created and manipulated at the unary set level, while the AP area is to be constructed and maintained by the binary association level. The AP area contains a collection of associative pointers. As discussed in section 4.1, we have made the distinction between associative pointers (AP's) and set pointers (SP's) since they represent different types of connections among data elements. An SP is used to chain BEUs of the same unary set together, while an AP is used to connect primitive elements of different primitive sets together.

Function types refer to the way elements of two binary associated sets are related. There are 4 types: 1:1, 1:n, n:1 and m:n. In this design, we have identified 3 different modes of binary association implementation:

1. Pointer mode: In this mode, 1:1 type is implemented through

inserting associative pointers into the AP area of the data element. An association pointer is the id of the counterpart element in this binary association. 1:n and n:1 are implemented by creating a subset. (Recall that in section 4.3.2 it was mentioned that there are two types of unary sets, one being the primary set, the other the subset, or secondary set.) Many-to-may type is effected through a dummy unary set that incorporates the binary elements involved. These structures are shown in Fig 4.9a to Fig 4.9c. Note that the subsets may be implemented as either linked lists or pointer arrays.

2. Physical duplication mode: Instead of storing the id of the assoicated element, the data part of that element is duplicated in the AP area, as shown in Fig 4.9d through Fig 4.9f.

3. Physical embedding mode: Under this scheme, the associated data element is physically stored within the associating element. This 'embedded' element may have its own identity, in the sense that it belongs to certain primary unary set and is recognized by the unary set processor, as shown in Fig 4.9g, or it may be a sub-unit, such that its manipulation always depends on manipulation of the embedding element, as shown in Fig 4.9h.

4.4.3 Data Definition Interface

Data definitions of primitive sets and binary relations are passed through this interface. Also passed are values of parameters in the parameter spaces to be discussed later. Two statements are identified:

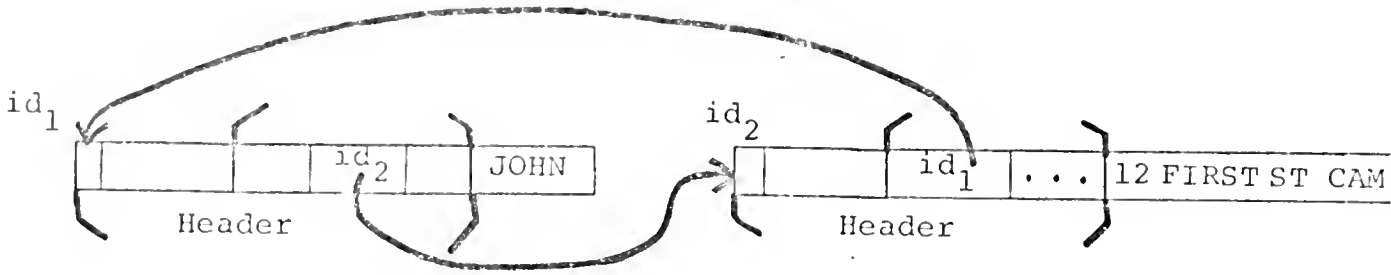


Fig 4.9a: Pointer mode, 1:1 Relationship

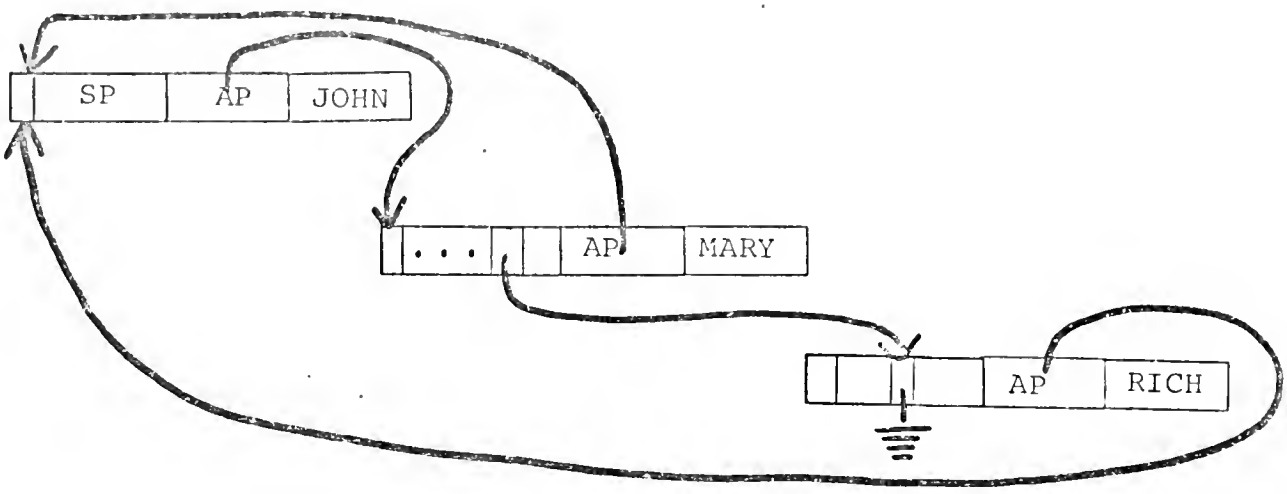


Fig 4.9b: Pointer mode, 1:n (children of JOHN) Relationship

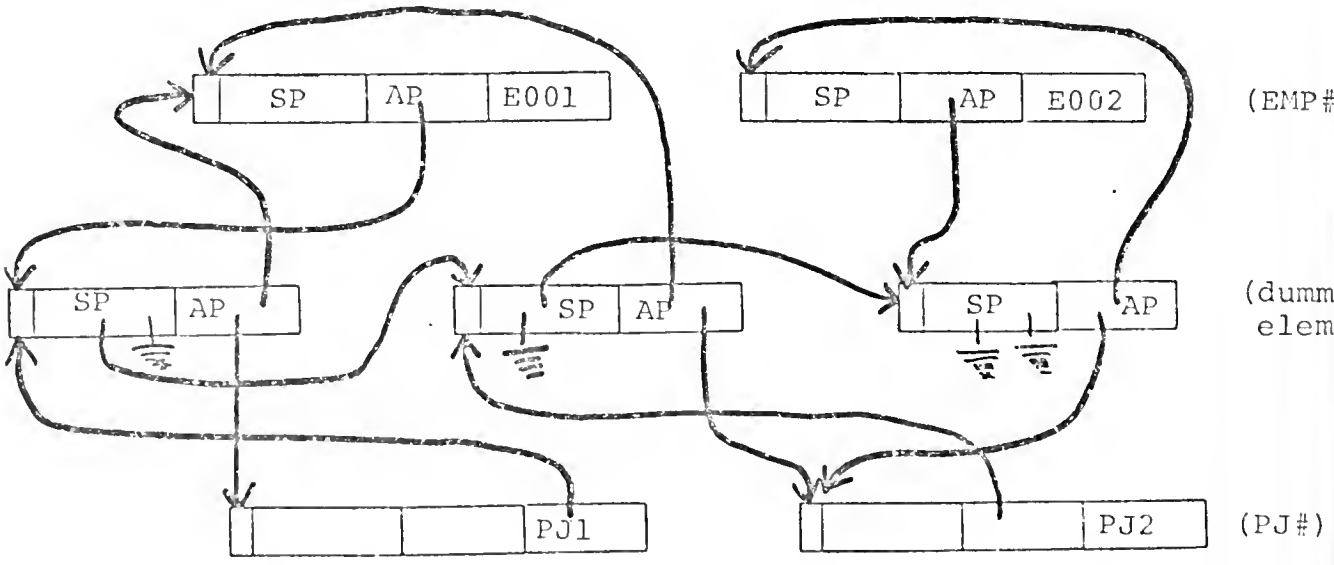


Fig 4.9c: Pointer mode, m:n Relationship (EMP_PROJ)

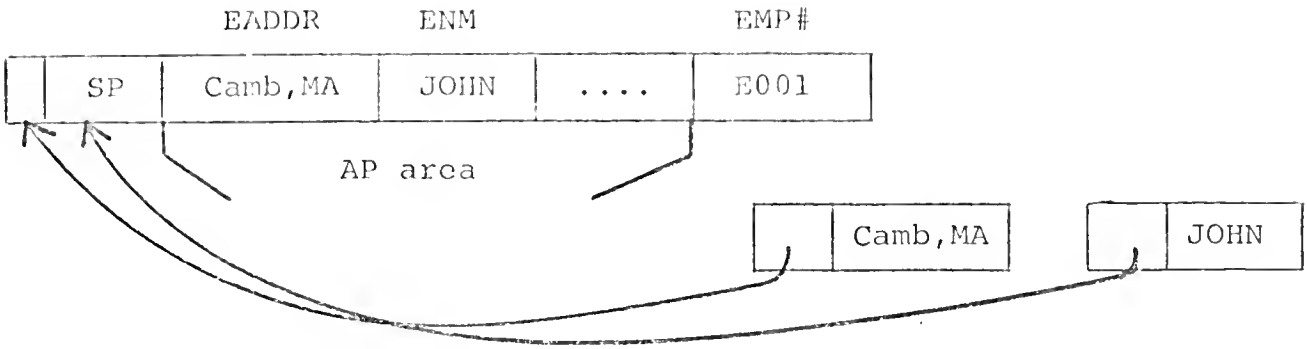


Fig 4.9d: Physical duplication mode; 1:1 Relationship

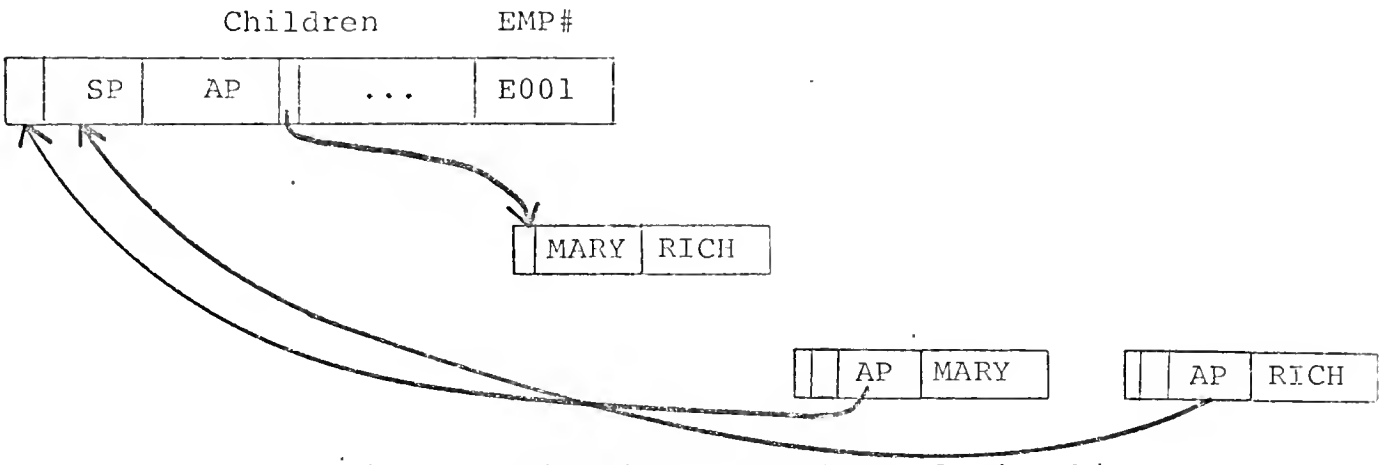


Fig 4.9e: Physical duplication mode; 1:n Relationship

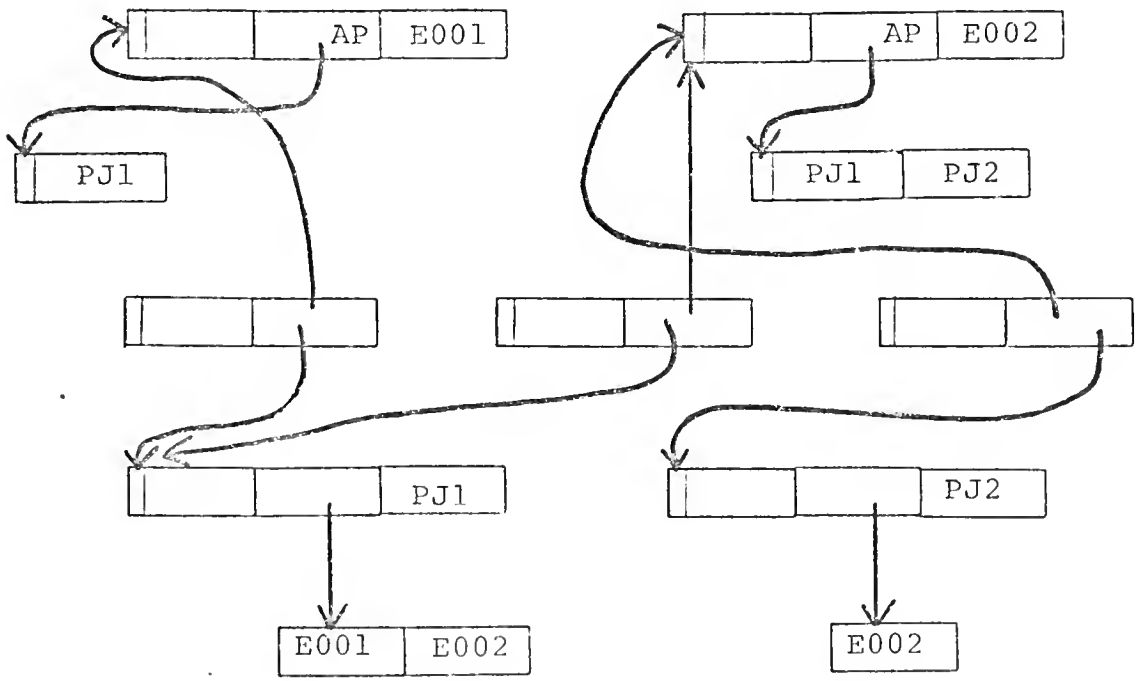
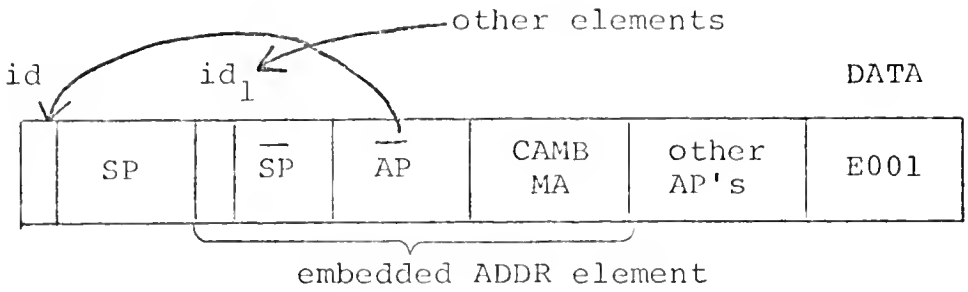


Fig 4.9f: Physical duplication mode; m:n Relationship



other address elements

Fig 4.9g: Embedding mode: embedded element has its own id

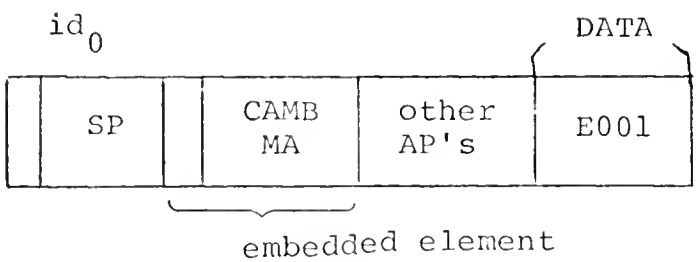


Fig 4.9h: Embedding mode: embedded element does not have its own id

Define_Pset (Pset_name, x∈SSPS, u∈ESPS)

Define_Bset (Bset_name, z∈ASPS)

ASPS represents the Association Structure Parameter Space, which specifies the function type, sets involved in this binary relation, and data structures chosen to implement this binary set. These implementation specifications provide guidance to the binary level in building the structure of the AP area of each element. The SSPS and ESPS are parameter spaces that are processed at lower levels (see sections 4.2.1 and 4.3.5). The binary level defines the unary sets, and therefore becomes aware of the existence of these unary sets. Brief statements of logic of these two definition commands are given in Fig 4.10.

Binary set definitions may be deleted. When a binary set is deleted, the primitive sets involved are left intact. On the other hand, when a primitive set is deleted, all binary sets defined upon it are deleted. Binary set definitions may also be modified. This modification may represent a data reorganization at the internal level. The detailed mechanism of this modification will be studied.

4.4.4 Operational Interface

Through this interface, insert, delete, update and retrieval of elements in primitive or binary sets are made.

Retrieval of the data base is done in two modes at this level: (1) Unique and (2) Set mode. Under the first mode, an item that satisfies the requirement is retrieved. Under the second mode, all items that satisfy the requirement are retrieved. To facilitate sequential processing, the first mode is further classified into self-contained commands and sequential operation commands.

Since the majority of existing database applications are still very procedure oriented, and not like the higher level query languages which are relatively self-contained, the sequential operation commands such as Get_Next become a necessity. When such a command is received at this level, the processor has the need to know what the current content of that variable is. For simplicity, it is assumed that such commands will pass information of this sort as part of the arguments, so that all commands through this interface will be self-contained. However, this assumption may be modified later for performance considerations.

At the instance level, deleting an instance of a binary relation affects only the association structure (e.g. pointers etc.) of the two elements, while deleting a primitive element deletes all binary relations stemmed from it. An integrity problem may occur when a primitive element is deleted. For example, if a primitive element is implemented by a BEU, then its id may be encoded into the BEUs of many related elements. If the id of the deleted element is not to be re-used, the problem is simplifed by setting a delete bit for the id. However, for efficiency, the id of a deleted element cannot, in practice, be left unused indefinitely. When the id of a deleted

element is re-used, those elements that are originally associated with the deleted element (and therefore have its id encoded into their AP areas) now would have a mistaken association to the new element that has assumed the id of the deleted element. One way to avoild this problem is to do as follows:

1. Locate the element to be deleted
2. Remove it from the primary set and all subset chains (i.e., update the link list of the unary connection)
3. Locate all other associated elements where the id of the element to be deleted is stored. Set it to null.
4. Delete the element (i.e., set the delete bit and return this id to the free slot chain of that page).

This integrity consideration also applies to situations where a data part is physically duplicated within another element. The implication of this consideration is that the database system has to have the capability to pull out elements that may be affected when a related element is changed. This may be accomplished, for example, with bi-directional vertical linked lists and bi-directional binary associations (i.e. a complete binary implementation).

The set of commands to be supported at this level is given in Fig 4.11.

4.4.5 Summary

This concludes our internal structure design. Higher level functions are now built on top of these structures, and communicate with the internal constructs through the interface provided at this level. We shall see at a later point how manipulation and query commands are eventually translated into operators that are accepted by internal constructs.

Fig 4.10: DD interface at the binary association level

Define_Pset(Pset_name, x ∈ SSPS, u ∈ ESPS)

logic:
1. format header according to the structure of CTLP.
2. Create_element('CTLP', header, data=Pset_name)
3. If this Pset is to be implemented as a Uset then
   Define_Uset(Uset_name, x,u)
4. return

Define_Bset (Bset_name, z ∈ ASPS)

This command defines a binary set. ASPS stands for Association Structure Parameter Space, which includes the following:
1. names and roles of the two primitive sets involved in this asso c.
2. function types (e.g. 1:n, n:1, 1:1, m:n)
3. storage mode (e.g. pointer, physical duplication or embedding)
4. the structure of the dummy set if functional type is m:n

logic:
1. check consistency of primitive sets involved in this binary set against CTLP
2. format z according to the structure of CTLB into Z
3. Insert_element ('CTLB', header=Z, data=Bset_name)
4. if function type not equal m:n, then go to 6, else
5. Define_Uset (Uset_name=set1/set2, x3, y3)
6. return

The following commands effect changes in the definitions of primitive and binary sets:

Update_Pset (Pset_name, changes in parameters)

Update_Bset (Bset_name, chagnes in parameters)

Delete_Pset (Pset_name)

Delete_Bset (Bset_name)

Fig 4.11: operational commands at binary association level

```
Create_p (Pset_name,[Byte_string] )
effects creation of an entry in the primitive set named here;
returns an id;


Create_b (Bset_name, id1,[data_of_role2(or id2)] )
effects creation of a binary association;
optionally returns an id;


Delete_p (Pset_name, id)


Delete_p (Pset_name, Rel_op, data)
effects deletion of all elements that satisfy
the predicate (rel_op,data) pair;


Delete_b (Bset_name, id1, [id2(or data2)] , [ER] )


Delete_b_multiple (id1, n, (Bset_namei, [idi(or datai)] , [ER] ,
    i=1,n))
     effects deletion of a binary association:
       If "ER" is not specified, then only the association
       between the two data is deleted;
       If "ER" is specified, then the association and the incident data
       are deleted;
       For one-to-many relationship, the incident data has to be
        specified; if not, all the associated data in that Bset_name
        are deleted;


Update_p (Pset_name, old_data(or id), new_data)


Update_b (Bset_name, id1, id2(or data2), new_data)
  updates the content of the incident data


Update_b (Bset_name, id1, id2(or data2), new_id)
  updates the association between id1 and id2 to be id1 and new_id
```

Retrieval of the database is done in two modes : unique, which retrieves one item at a time, and set, which retrieves a set of elements. A limited ability to process predicate requirements is incorporated.

unique mode:

Find (Pset_name, data, fnd*)

Select_p_first (Pset_name, data*, id*)

Select_p_next (Pset_name, id0, data*, id*)
retrieves the element that follows element at id0;

Select_b (Bset_name, data1(or id1), data2*, id*)
Retrieves the binary associated element of data1 or id1 in Bset_name;

Select_b_first (Bset_name, data1(or id1), data2*, id*)
retrieves the first occurrence of binary associated element of data1(or id1) in bset_name;

Select_b_next (Bset_name, data1(or id1), id0, data2*, id*)
retrieves the next occurrence of the binary associated element after id0 of data1 in bset_name; (note: meaningful only when Bset_name is 1-to-many or many-to many)

Select_b_join_first (m, (Bset_namei, datai(or idi), i=1,m))
gives the first element that satisfies m binary predicates;

Select_b_join_next (m, (Bset_namei, datai(or idi), i=1,m), id0)
gives the next element that satisfies the m binary prdeicates next to id0;

set mode:

Retrieve_p_set (Pset_name, n*, ptr*)
gets the set of element in Pset_name; n is the number of element returned; ptr points to the area where the whole set can be found;

Retrieve_b_set (Bset_name, n*, ptr*)

Select_p_set (Pset_name, rel_op1, data1, n*, ptr*)
gives the set of element that satisfies the predicate;

Select_b_set (Bset_name, rel_op, data1, n*, ptr*)
gives the set of binary relation that satisfies the predicate;

Select_b_join_set (n, (Bset_namei, Rel_opi, datai(or idi), i=1,n))
  gives the set of the elements that satisfy the m binary prdicates;

Count_p_set (Pset_name, n*)
     gives the number of elements in the primitive set;

Count_b_set (Bset_name, n*)
     counts the number of binary 'tuples' in the binary set;

V. DATABASE SEMANTICS

Making use of functions provided by the internal structure levels, 3 additional levels are established to build semantic constructs. These levels enrich the data model and provide a facility for expressing schema constraints.

5.1 N-ary Level

5.1.1 Introduction

The N-ary level processes constructs defined in the conceptual schema. The conceptual schema defines the total, integrated 'view' of the database with clean semantics. The major functions of this level are the following:

(1) Accept the data definitions of the conceptual schema expressed in the BN model (as shown in Fig 2.6), check for consistency and build the conceptual schema catalogue.

(2) Process operations on instances of the conceptual schema.

5.1.2 N-ary Data Definitions

The DD interface consists of the following definition statements:

Define_Vset (Vset_name, other parameters)

Define_Nset (Nset_name, (attribute_name, y∈ATPS)_list)

where Vset stands for Value Set, Nset for Entity Set, and ATPS stands for Attribute Parameter Space.

The detailed format of data definitions processed at this level has been discussed in chapter 2 and shown in Fig 2.6. Either Define_Vset or Define_Nset will result in a node being created in the schema, and each attribute defined in the Define_Nset statement will result in an arc being created. Consistency checking includes such items as proper equivalence definition, proper domain definition, and compatible syntax and semantic specification on the arcs. Processing of a set of data definition statements results in building a set of catalogues to be used by the N-ary level in interpreting and executing operations on the database. There are two catalogues to be maintained: Primitive and Binary. The primitive catalogue contains an entry for each node defined in the schema, and the binary catalogue contains an entry for each direction of an arc defined. The catalogues are built in such a way so as to facilitate cross referencing.

5.1.3 N-ary Operators

In order to distinguish between instances of Nsets defined at this level and instances of constructs defined at external view levels, the former is called an entity record from now on. Operators to retrieve and update the entity records are supported. Again, there are two modes of retrieval: set mode and unique mode. Set mode will give all

the entity records that satisfy the requirement, while unique will give just one. Sequential operator Retrieve_next is also included.

The retrieval operators are listed below:

Retrieve_Set (Nset_name, attribute_name_list) WHERE (attribute_name, rel_opr, value)_list

Retrieve_Unique (Nset_name, attribute_name_list) WHERE (attribute_name, rel_opr, value)_list

Retrieve_next (Nset_name, attribute_name_list) WHERE (attribute_name, rel_opr, value)_list CURRENT IS ((attribute_name, value)_list)

where rel_opr are the relational operators such as =, >=, >, etc. Also note that if the 'next' mode is specified, the current content of the attribute_name list has to be supplied. If the domain of any of the attribute_name in the list is not a value set, the attributes of that non-value attribute to be retrieved are specified in the command.

To update the instances of an entity set, one has to be careful in defining the meaning of the update. (Recall that Nsets defined here are not restricted to normalized relations). Therefore, the following rules are imposed:

1. To insert a new entity record (e.g., to add a newly hired employee's record into the Nset EMPLOYEE), all the immediate attributes may be passed. Values of a 1:n or m:n attribute have to be enclosed in parantheses. An example is given in Fig 5.1.
2. To delete an entity record (e.g. to delete the record of an

Suppose we have an n-ary entity set EMP defined as

```
EMP(EMP#,ENAME,DEPT,CHILDREN);
```

To create a new employee "Joe John", use the following INSERT_ENTITY statement:

```
INSERT_ENTITY (EMP, EMP#="0907",
               ENAME="JOE JOHN",
               DEPT="0015",
               CHILDREN=("MARY JOHN", "RICH JOHN"))
```

To add another child to Joe John's CHILDREN attribute, use the following INSERT_ATTRIBUTE statement:

```
INSERT_ATTRIBUTE (EMP, ENAME="JOE JOHN",
                  CHILDREN=("JEFF JOHN"))
```

Fig 5.1 and 5.2: INSERT_ENTITY and INSERT_ATTRIBUTE

employee who has quit from the Nset EMPLOYEE only one of the candidate keys has to be passed. All the rest is done automatically.

3. To insert or delete an instance of a 1:n or m:n attribute (e.g., to insert a new child into the attribute CHILDREN of the relation EMP), a candidate key of the entity record and the attribute instance are given. All other attributes within that relation are not cited. See Fig 5.2.

5. Updating of any attribute value is done pair-wise, i.e., only a candidate key of the entity record and the attribute to be changed are involved each time.

The operators to be supported for update are listed below:

Insert_Entity (Nset_name, key value, (attribute,value)_list)

Insert_Attribute (Nset_name, key value, (attribute value(s))_list)

Delete_entity (Nset_name, key value)

Delete_Attribute (Nset_name, key value, (attribute, value(s))_list)

Update (Nset_name, key value, (attribute, old value, new value))

The logic of these operators is briefly discussed in the next section. The detailed algorithms as well as the effect of these operations will be studied.

5.1.4 Retrieval Strategy

When retrieval commands for entity records are given, they are translated into operators upon underlying binary associations. These binary operators are then given to the next level across the binary association interface. This translation procedure involves access path selection. Since the attributes of an entity set are implemented as binary asociations to the entity node, the simplest solution is to establish the instance of the entity node first, and then follow the binary associations to obtain its attributes.

There may be different ways to establish the desired instance of an entity node. Therefore, optimization of the construction effort is to be considered. In general, a record can be constructed in many ways, depending on the choice of the starting role and the path to be followed. To clarify this idea, imagine that the mapping of an n-ary set to its binary form is to be stored as a tree, where attributes are represented by nodes. Instances of attributes at higher levels of the tree are to be established before those of the lower levels of the tree. Branches represent binary associations in consideration. Then a natural choice of the tree from of an example retrieval command is shown in Fig 5.3. However, there are equivalent trees that may represent exactly the same relation with a different starting role. An example is given in Fig 5.4. It is seen that the latter graph 'hangs' the tree in a slightly different orientation and completely changes the path of construction.

Since the shape of the tree may affect record construction performance, it is a decision to be made intelligently by a retrieval_strategy module. The issue may even involve the internal

structure, such as indices or sorted lists. To see what all these mean, we use the above example to illustrate the different construction efforts that may be involved in these two trees. In the example, suppose PROJ# is specified in the retrieval command to be the sequence field. Ideally the system should translate the command into a procedure such that records are constructed according to the sequence order. Alternatively, it may sort the records after they are entirely generated by subjecting them to a sort module. If we choose the second tree form and, if the entity set 'PROJ#' is sorted in the internal structure, then records will be generated in the desired order automatically. On the other hand, the first tree form will not result in a set of records ordered by PROJ#, therefore requires a further sort.

Another more obvious consideration would be that, if values of some attributes are to be restricted (e.g. PROJECT='system'), it may save some effort if construction starts at the restricted attribute. Restriction usually takes place when the WHERE clause is used in the command.

To summarize, the strategy used in this design is listed below; more examples are also given:

(1) If no predicate is given , the entity record construction starts from one of the candidate key attributes of the entity set. (Fig 5.5a).

(2) If a key attribute of the entity is restricted in the predicate, the entity record(s) is(are) constructed starting from

that key. (Fig 5.5b).

(3) If none of the key attributes are restricted in the predicate, while a non-key attribute is, then there are two approaches:

(a) Locate those unary instances of the restricted non-key attribute that satisfy the restriction and trace back to its entity node instance (Fig 5.5c).

(b) Visit every entity in the set. Start from some key attribute and collect all relevant fields, and then see whether this entity record satisfies the predicate. Do this for every entity in the set. (Fig 5.5d).

(4) If multiple non-key attributes are specified in the predicate, the choice again would be either 3(a) or 3(b) or some combination of the two.

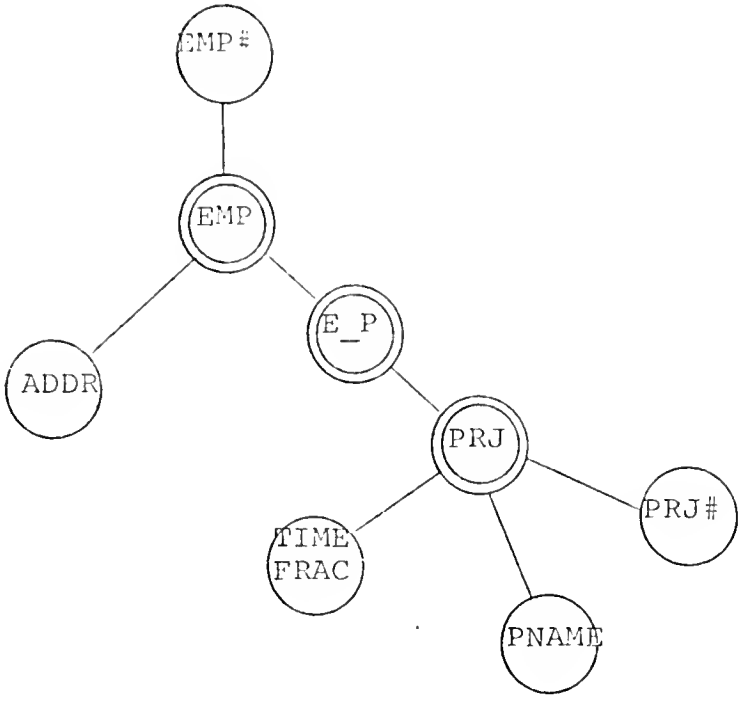
(5) If sequence attributes are specifed, then there are again two approaches:

(a) Construct entity records according to the sort order

(b) Construct entity records with another strategy and sort them at the end.

To centralize this issue of retrieval strategy, a retreival_strategy module is singled out. Conceivably, this module can take advantage of developments in the area of query-decomposition. (for example, <Yao79, Astrahan76>.)

When a storage operation is given (e.g., insert or delete), the n-ary level has to carry out its semantic ramifications. For example, in inserting a new entity instance into an Nset, those attributes that are declared to be total (i.e., those that cannot have value



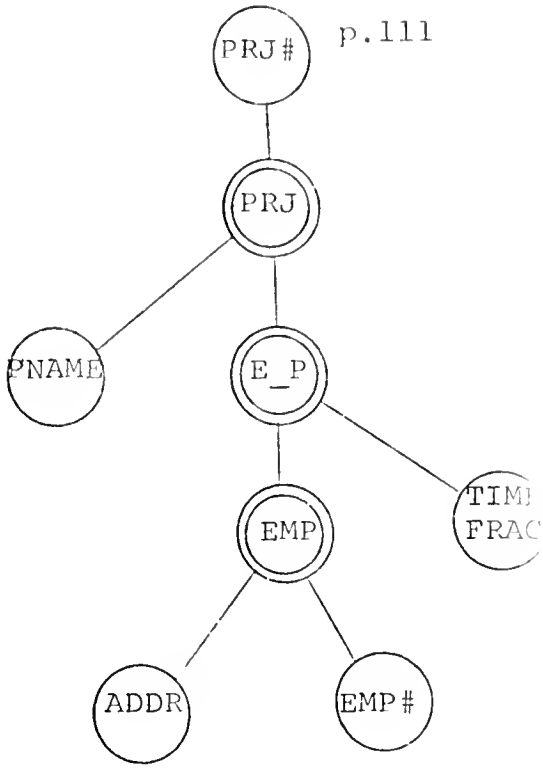


Fig 5.3 & 5.4: Tree forms of access path selection

command: RETRIEVE_SET (N.EMP,(EMP#,ENAME,DEPT(DEPT#,DNAME)))

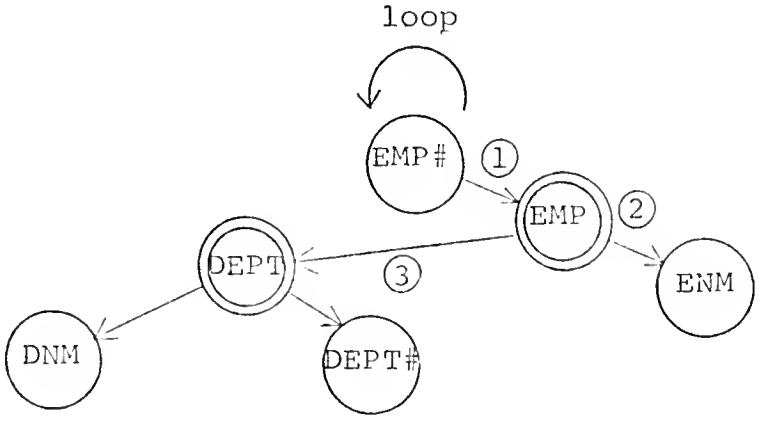


Fig 5.5a: Retrieval strategy (1)

command: RETRIEVE_UNIQUE (N.EMP,(EMP#,ENAME,DEPT#)) where (EMP# = '0909')

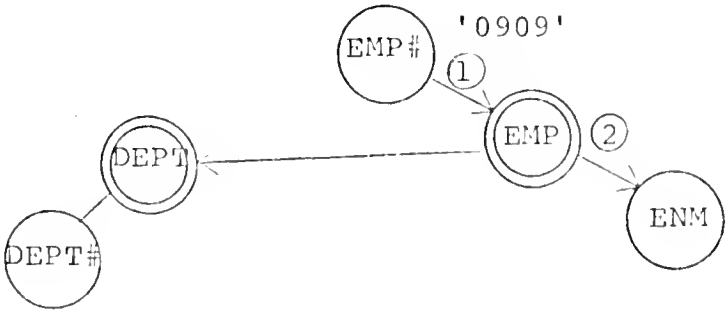


Fig 5.5b: Retrieval strategy (2)

command: RETRIEVE_SET (N.EMP,(ENAME,ADDR)) where (DEPT(DNAME = 'SYSTEM'))

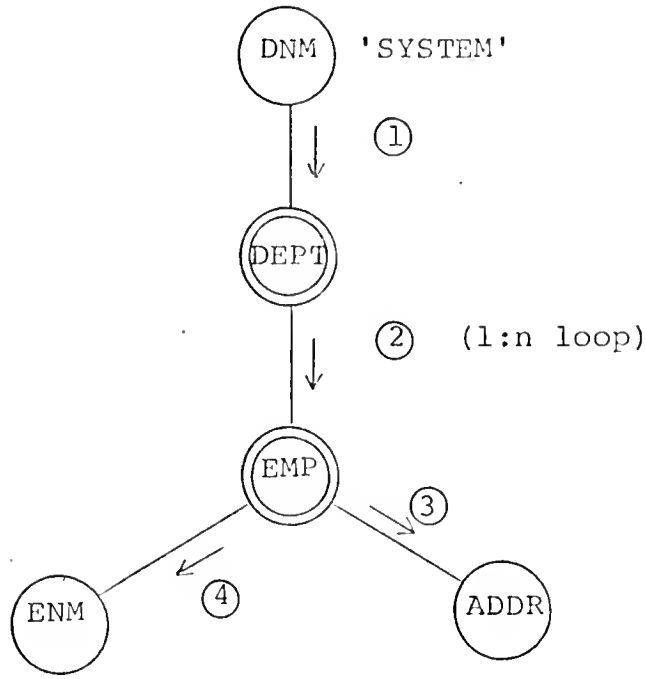


Fig 5.5c: Retrieval strategy 3(a)

command: RETRIEVE_SET (N.EMP,(ENAME,ADDR)) where (DEPT(DNAME = 'SYSTEM'))

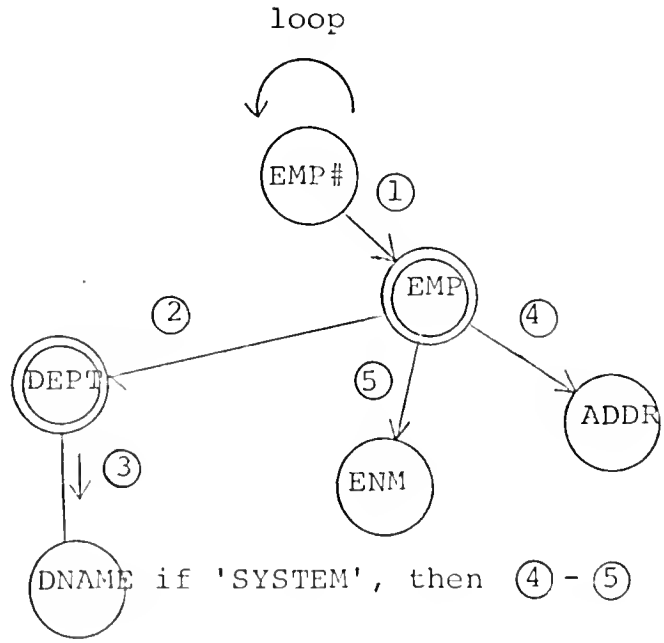


Fig 5.5d: Retrieval strategy 3(b)

'undefined') must be supplied. Also, if it is an instance of a hierarchical arc, an instance of its hierarchical 'parent' must exist. As another example, when an instance of an entity is deleted, all the associations to other instances have to be deleted; and if it is a hierarchical parent to some other node, the instance of this hierarchical 'child' must be deleted; and so on. These rules must be implemented to make sure that database assertions are maintained. (At the virtual information level and the data validity level to be discussed later in this chapter certain database assertions that are not carried out at this level are also maintained.)

5.1.5 Entity record construction

We now show by some examples how an entity record is constructed. The format of the Nset catalogue and data structure used during construction are also exemplified.

Suppose we have a database composed of 3 entity definitions as shown in Fig 5.2c. An example format of the Nset catalogue is given in Fig 5.6. Then processing the following retrieval command:

Retrieve_Set (N.EMP, (EMP#, ADDR, EMP_PROF (PROJ (PROJ#, PNAME),TIMEFRAC)))

is equivalent to generating a table of a format shown in Fig 5.7a. This command is first passed to the retrieval strategy module, which evaluates the possible access paths. A possible access path is depicted in Fig 5.7b. It shows a tree diagram that portrays the route according to which the database is to be traversed in order to carry

out the command. A data structure that corresponds to this tree is then generated by the retrieval strategy module. An example of this structure is shown in Fig 5.7c. The general logic of record construction given this data structure is shown in Fig 5.8a, while the procedure used to carry out this example retrieval command is shown in Fig 5.8b.

Nset_Cagalogue

Nset_name	No_of_attr	Ptr_to_attr_list
EMP	3	
PROJ	3	
EMP_PROJ	2	

Attr_List

Attr_name	Key	Domain Type.name	Attr_name (Op_name)	Syntax	Sem
EMP#	YES	V.EMP#		1:1	etc.
ADDR		V.ADDR		1:1	
EMP_PROJ		N.EMP_PROJ		1:n	

Fig 5.6 An example of the format of the Nset catalogue

Field_no	1	2	3	4	5
Field_name	EMP#	ADDR	PRJ#	PNAME	TIMF
(example)	0907 0908	Camb Lex	015 015	System System	20% 50%

Fig 5.7a: A table to be generated by a retrieval command

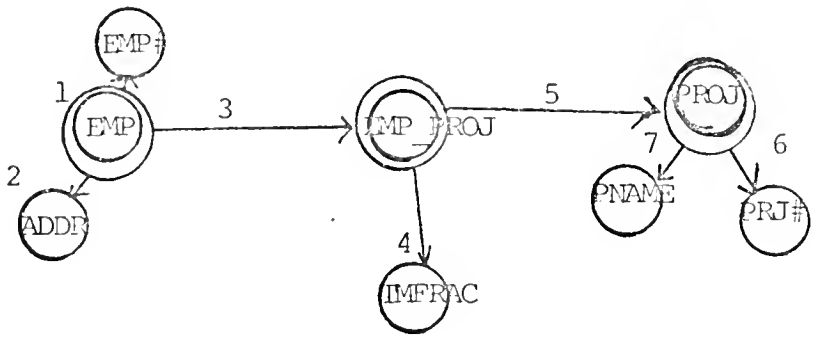


Fig 5.7b: A retrieval strategy for command in fig 5.7(a)

NO	DOMAIN	FIELD_NO	BSET_NAME	PREC	TYPE
1	N.EMP	T1	-	-	-
2	V.EMP#	1	EMP.EMP#	1	1:1
3	V.ADDR	2	EMP.ADDR	1	1:1
4	N.EMP_PROJ	T2	EMP.EMP_PROJ	1	1:n
5	V.TIMF	5	EMP_PROJ.TIMF	4	1:1
6	N.PROJ	T3	EMP_PROJ.PROJ	4	n:1
7	V.PROJ#	3	PROJ.PROJ#	6	1:1
8	V.PNM	4	PROJ.PNM	6	1:1

Fig 5.7c: Data structure generated by the retrieval strategy module -- an example of fig 5.7(b)

```
Record_Construction (Catalogue, Record_map)

Begin:  Reserve working space WS, ID for the record, and stack ST;

Initiate:  /*Initiate the first record */
    N=number of nodes to be traversed;
    I=1;
    J=Field_no (I);
    (WS(J),ID(J))=Select_e_first (Domain(I));
    NEXT=2;
    Call Rest_record;
    Call Print_record;

Continue: /*Process other records*/
    Do until EOD_of_starting_node;
       Do while stack not empty;   /*checking stack for looping*/
          I=ST(SP);
          J=Field_no(I);
          (WS(J),ID(J)))=Select_b_next (Bset(I), WS(Pred), id_o=ID(J));
          If not end_of_data then do;
             NEXT=I+1;
             Call Rest_record;
             Call Print_record;
             end;
          Else pop stack;
       end;
       /* Establish next data of starting node*/
       I=1;
       J=Field_no(I);
       (WS(J),ID(J)))=Select_e_next (Domain(I),id_o=ID(J));
       NEXT=2;
       Call Rest_record;
       Call Print_record;
    end;
    Return;

Rest_record: /*Internal routine for completing a record*/
    Do I=NEXT to N;
    J=Field_no(I);
    If Bset(I) is 1:1 or n:1 then
         (WS(J),ID(J))=Select_b(Bset(I),WS(Pred));
    Else do;
       Push I on stack;
       (WS(J),ID(J))=Select_b_first (Bset(I), WS(Pred));
       end;
    end;

Print_record: /*Internal routine for output the record just constructed*/
    Print WS;

End Record_Construction;
```

Fig 5.8a: A General Logic for Record Construction
(Given schema catalogue and record map produced by Retrieval_module)

For example, the following procedure will be executed by the record-construction module when invoked to build the table shown in Fig 4.7a:

```
1    /*Reserve WS, ID and ST as follows:*/
```

	1	2	3	4	5	T1	T2	T3
WS	EMP#	ADDR	PROJ#	PNAME	TIMEFRAC	N.EMP	N.E_J	N.PRJ
ID								

ST: sp →

```
2    /*Establish the first record:  starting node:*/
                (ID(T1),WS(T1) = SELECT_E_FIRST(N.EMP)
2.1      /*Next node:*/ (ID(1),EMP#) = SELECT_B(EMP.EMP#,WS(T1))
         /*Next node:*/ (ID(2),ADDR) = SELECT_B(EMP.ADDR,WS(T1))
         /*Next node is E_J, which has one-to-many relationship
           with N.EMP a stack entry is made to control the loop*/
           ST(SP) = 4
               (ID(T2),WS(T2)) = SELECT_B_FIRST(EMP.E_J,WS(T1))
         /*Next node:*/ (ID(5),TIMEFRAC) = SELECT_B(E_J.TMFR,WS(T2))
         /*Next node:*/ (ID(T3),WS(T3)) = SELECT_B(E_J.PRJ,WS(T2))
         /*Next node:*/ (ID(3),PROJ#) = SELECT_B(PRJ.PRJ#,WS(T3))
         /*Next node:*/ (ID(4),PNAME) = SELECT_B(PRJ.PNM,WS(T3))

3    /*The first record is completed; It is printed; now the next
       record:
3.1    Since the stack is not empty, we start from the node to be
       looped, which is the 4th node:*/
               I = 4; J = T2;
               (WS(T2),ID(T2)) = SELECT_B_NEXT(EMP.E_J,WS(T1))
               /*and establish the rest of the record;
                 continue doing this until end of EMP.E_J is
                 encountered*/

4    /*Pop stack; now that the stack is empty, establish the next data
       data of the first node:*/
               (ID(T1),WS(T1)) = SELECT_E_NEXT(N.EMP, id_o = ID(T1))
         /*Then complete the rest of the record as in 2.1 and
           check stack as in 3.1;

5    Stop when no more EMP# is available*/
```

Fig 5.8b: An example of record construction

5.2 Virtual Information Level

5.2.1 Introduction

Semantic or statistical relationships among elements are often prevalent in a database. For example, an employee's age can be derived from his birthday and the current date; the accrued interest of a bank account is equal to its balance multiplied by the interest rate. If the derived element is also stored, certain consistency problems may occur.

There are two approaches to the issue of database accuracy. One is to maintain a catalogue of consistency constraints as well as some "housekeeping" routines that traverse the database periodically to enforce satisfaction of these constraints. Housekeeping routines may also be invoked when a sensitive data element is to be updated.

Alternatively the DBMS may maintain database consistency by eliminating from the stored database those data fields that can be derived from other data. It maintains a catalogue of functions to be used in the derivation process. This gives rise to the term 'virtual information', signifying information that is not physically stored but may be computed. This approach may do away with those housekeeping routines used in the previous method. However, it adds to the overhead of computing and recomputing a data field whenever it is accessed. Also, since they are not physically stored, it is difficult to make direct retrieval against these fields. For example, a query to get all the accounts that have accrued interests exceeding a certain level

would require that the accrued interest of every account be computed. The database designer has to take tradeoffs of these approaches into consideration in making internal structure decisions.

Extending this concept, any information may be 'derived' from combinations of algorithms and data that is physically stored <Madnick73>. On one extreme, the information may be derived purely through algorithms (e.g. Sine and Consine functions). On the other extreme, information may be derived through a direct search in the database (e.g. an employee name given his employee number). In between these extremes, however, there is information that is derived through a combination of algorithms and data (e.g., a query on a person's age is derived by retrieving the current date and his birthdate from the stored database and then performing a subtraction). Under this framework, several categories of virtual information are identified here:

(1) Computed facts: algorithms to compute from stored data;

(2) Representation: data type conversion functions;

(3) Encoding: data string encoding functions.

5.2.2 The General Mechanism

To support the virtual information implementation, this level serves as a front gate to the level immediately lower to it, namely, the n-ary entity level.

It keeps a catalogue of all information that is virtually defined. Every request to create, retrieve, update or delete a unit of data is filtered through the gate, and if any virtual information is involved in the request, it provides functions for the transformation of data.

(1) Computed facts:

Any attribute definition that has its virtual flag (VT) on is extracted and processed at this level without being passed down to the next level. The virtual attribute is entered into a catalogue. The derivation function is also stored along with (or chained to) the catalogue entry. The function may use both raw data and virtual data in its algorithm, therefore effects nested virtual information. For example, the attribute VOLUME is derived from AREA and HEIGHT, while AREA may in turn be derived from LENGTH and WIDTH.

(2) Representation:

As a request for records is given, the data type of each field is also given. If it is different from the stored data type, transformation is done before the element is passed down to the storage or returned to the requester. Therefore, the virtual information level has to be aware of the data types of all the stored elements.

(3) Encoding

Attributes that need to be encoded before they are passed down are noted at this level. Encoding functions are given. This type of

virtual information service is especially useful for encoding data definition languages (e.g. relation names) before they are processed by lower levels.

5.2.3 Data Definition Interface

The Data Definition statements of database constructs (Primitive, Binary or N-ary) that have a virtual flag attached to are intercepted and processed at this level. The definition statement looks like the following:

Define_Nset (Nset_name, (attribute_name, y∈ATPS, v∈VTPS) list)

where ATPS is recognized and processed by lower levels, while VTPS represents Virtual Parameter Space, and v is decoded at this level; if appropriate, entries are made in the virtual information catalogue accordingly.

5.2.4 Operational Interface

The operators are almost identical to those at the n-ary associtions level. Operations on nonvirtural attributes are passed untouched to the lower level, while operations on virtual attributes are intercepted. Encoding of database mnemonics such as Nset or Bset names is also accomplished at this point. An elaborate catalogue to record these mnemonic designations is to be maintained.

While the derivation function of a virtual attribute can be changed by updating the attribute definition, the individual virtual element cannot be updated. (For example, request to update JOHN's age is to be considered meaningless). It also follows that individual virtual element cannot be inserted or deleted; only the virtual catalogue entry may be thus operated on. Therefore, only retrieval operations are to be processed at this level.

5.3 Data Validity Level

The conceptual schema may include constraints on the set of legitimate values some data elements in the database are allowed to assume. For example, elements describing a date has to be confined to 12 months and 28 to 31 days in the calender; employee's salary data may not be allowed to exceed a certain maximum; etc. These constraints uphold the data validity of the database.

Another type of semantic restriction affects the interrelationship among data elements. For example, a student cannot be allowed to take courses that have conflicting schedules, or be appointed a teaching assistant before he clears all the 'incompletes' from the last term. Another example would be that the total budget of a year should be equal to the sum of all the allocated budgets of the subordinate branches. (Note that if a functional (i.e. exactly computable from one another) relationship exists between data elements, the information that may be derived needs not be stored, but computed when requested. This is the task of the virtual information level mentioned in the previous section). When these constraints are specified, the DBMS has to translate them into routines that monitor and enforce the realization of them. Some descriptions of the integrity constraints are presented in <Eswaran75>.

There are also situations that are allowed to occur, but users are to be warned or alerted immediately <Morgan75>. A sudden increase in the death rate at a certain region detected by a system used to monitor the health management would be an example. The DBMS is expected to

output the alerting message when 'abnormal' conditions such as these are detected in the databse.

The data validity, integrity and alerting problems are targets of the current level.

5.3.1 General Mechanism

All primitive, binary or nary sets of the database that have these constraints attached are flagged when they are defined. The constraints, written as part of the data definition, is to be translated into a procedure that is to be executed at this level, mostly when update of elements of these sensitive sets are encountered. Operators implemented at levels below shall be used in the translated procedures. Constraints may also be specified for virtual attributes.

There are two modes of enforcement: (1) periodic checking and (2) checking invoked only when data elemnts are updated. In implementing the first mode, a hardware timer has to be made available. An event list, sorted by scheduled time of the events is maintained and checked constantly, either by way of timer interrupt or through the use of a dedicated processor. If the time of a scheduled event has arrived and the routine is invoked the next scheduled time for this routine is inserted into the event list. A message is printed when a database error is detected, and the error condition may be logged or left pending for correction.

To support the second mode, this level maintains a catalogue of sensitive database constructs. Any update (including insert and delete) of an element of these sets has to be filtered through the procedures that 1) precompute the result, 2) check for legitimacy of the result, 3) determine whether to accept or deny the request, and 4) if a request to be denied, return an explanatory message. For operations on non-sensitive data, the requests are passed directly to the next level.

How to select the mode of constraint implementation is, again, a DB designer decision, which has to have situational factors taken into considerations.

The simplest type of constraint would be the data type, i.e., whether it must be numeric or alphanumeric. More strict constraints may be specified in terms of 1) the range of values allowed for a continuous data element or 2) a catalogue that contains a set of legitimate values of a discrete data element. When inter-relationship is a consideration, the procedure has to retrieve other data in the database in order to do the computation.

5.3.2 DD Interface

The most complex problem at this level is the data definition interface where by the specification of the validity and consistency constraints are crossed. Commands are also available to make changes to these constraints.

Define_Vset (Vset_name, other parameters, w VLPS) or Define_Nset (Nset_name, (attribute_name, w VLPS, v VTPS, y ATPS) list)

where VLPS stands for VaLidity Parameter Space, to be intercepted and interpreted at current level, while parameters in the rest parameter spaces are passed down. (See section 5.1.2 and 5.2.3 for explanation).

5.3.3 Operational Interface

Commands to manipulate data elements are passed through. The operators are identical to those to be accepted by the next level, and checking is performed for sensitive data sets.

VI. USER VIEWS AND DATABASE SECURITY

6.1 Introduction

The conceptual schema designer may have structured the information into entities and attributes. Operators to operate upon these constructs are implemented in the levels below. End users of the database, however, may see a structure of the data that is different from that of the conceptual data model. In particular, they may want to look at the data as organized into a different set of relations or a hierarchy of segments. They then choose various sublanguages accordingly to operate on these "external constructs". In this manner, the end user is allowed to look at data in a way most natural to his application and choose a data sublanguage that he feels most comfortable to use. In addition, existing application programs are protected from becoming obsolete due to changes in data structure. These different ways of looking at the structure of the database constitute different "user views", also called "external views", of the database. To support external views, these views have to be defined and mapped onto conceptual structures. Operators performed upon external views have to be translated into operators upon entity sets defined at the conceptual level.

In addition to providing user views, the DBMS is also required to maintain database security. Database security refers to the control of access to the database. Since an integrated database is accessed by a number of users, and since each of them may be granted permission to

only part of the database, the DBMS has to have the capability to identify a user, determine his access limitations, and accept or reject an access request. Furthermore, permissions may be differentiated in terms of Read and Write. There are generally two approaches to maintaining database security control <Hsiao79>. The first one is by way of view definitions, and the second one is through query modification. Here we choose to use the first approach.

We have identified a three-level hierarchical structure for handling user views and security control. At the top, the View Authorization Level authentiates a log-on user and authorizes the user to a particular view. Next, the View Translation Level keeps track of mappings between constructs defined in external views and the conceptual schema, and translates the operators. Below it, the View Enforcement Level checks for legality of the operation of a view.

Before going into these individual levels, a formal definition of mapping between external constructs and conceptual constructs is given in the next subsection. To illustrate transformations in a coherent fashion, a single database example is used throughout this chapter. The conceptual definition of this example database is shown in Fig 6.1.1.

6.1.1 Mappings

Mapping herein refers to the correspondence between an external scehma and the conceptual schema. One of the reasons why we choose to

DATABASE EMP_ASSIGNMENT

CONCEPTUAL DEFINITION:

EMP(E#, EN,ADDR,DEPT,E_P, JB_HSTRY,MGR_OF,PRJ=E_P(PRJ))
DEPT(D#,DN,LOC,EMP)
PRJ(P#,PN,MGR,EMP=E_P(EMP),E_P)
E_P(EMP,PRJ,TIMEFRAC)
J_H(EMP,JOB,DATE)
JOB(JB#,JBN,JB_HSTRY)

Primitive/Binary Data Structure Diagram of EMP_ASSIGNMENT:

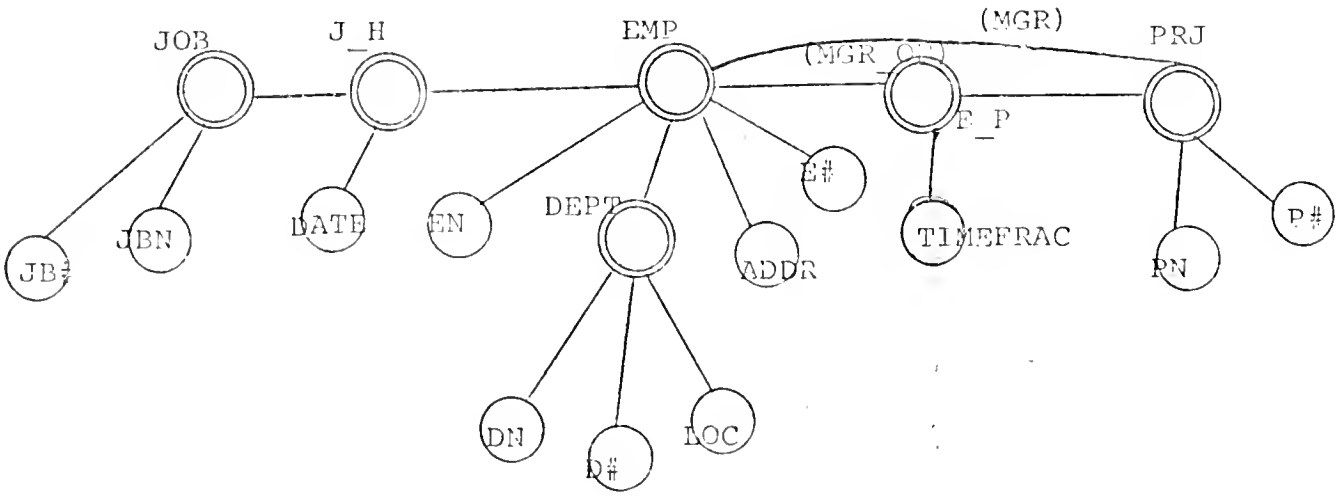


Fig 6.1.1: An example database

use binary relations as the underlying conceptual structure, as mentioned earlier in section 2.2, is for its flexibility in transformation (or "deconceptualization"). In our system, mapping is provided for constructs in three different external data models, or "types of views", namely, the relational, the hierarchical, and the network data models. Constructs in these models are mapped using a common mapping language which describes any external construct in terms of the following conceptual constructs:

(1) Entities and their direct attributes or derived attributes;

(2) Binary relations between entities and/or attributes;

(3) Predicates restricting above constructs;

In essence, any external construct (e.g., a relation or a segment) is to be mapped to a portion of the integrated database described by the conceptual schema, and the mapping statements specify this "portion". The BNF specificatin of this mapping language is given in Fig 6.1.2a. Some examples of mapping language statements and their corresponding graphical representation, called the tree form of the mapping, which is basically a "clipping" of the data structure diagram of the integrated conceptual schema, are given in Fig 6.1.2b.

```
Mapping_statement ::= Entity_statement | Binary_statement
Entity_statement  ::= Entity_attribute_clause, predicate clause
Entity_attribute_clause ::= Nset_name (attr_list)
attr_list         ::= attr_phrase | attr_list attr_phrase
attr_phrase       ::= attr_name | attr_name(attr_phrase)
Binary_statement  ::= Nset_name (attr_phrase)
predicate_clause  ::= (condition_list)
condition_list    ::= condition | condition_list condition
condition*        ::= attr_phrase, comp_op, target_phrase
comp_op           ::= >|=|<|>=|<=|¬=|∈|∋
target_phrase     ::= variable_name | literal
```

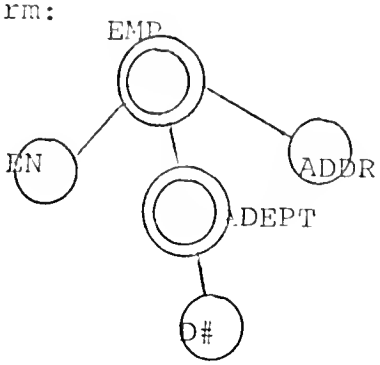
*: Conditions can be further elaborated to include set-theoretic comparisons.

Fig 6.1.2a: A BNF specification of mapping language

(1) External construct e_1 mapped to unrestricted entities and attributes:

e_1 = EMP(EN,ADDR,DEPT(D#))

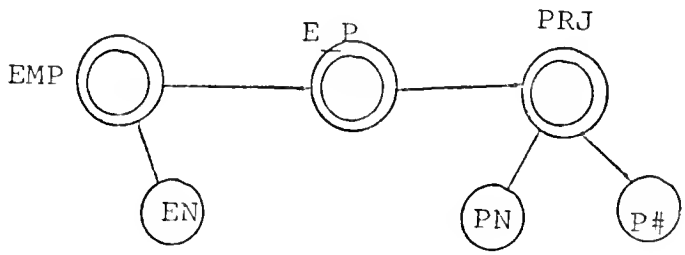
treeform:



(2) e_2 mapped to restricted entities and attributes:

e_2 = PRJ(PN,P#) WHERE (EMP(EN)=Var.EN)

treeform:



(3) e_3 mapped to a binary relation

e_3 = EMP(JB_HSTRY)

treeform:

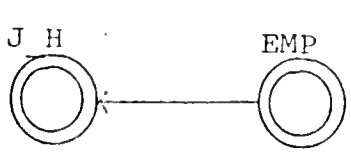


Fig 6.1.2b Mapping statements and their 'treeform' graph

6.2 View Enforcement Level

6.2.1 Introduction

This level integrates and coordinates all external views. It performs two major tasks: (1) process operational security parameters of the views and check for compatibility of these parameters among all views; and (2) enforce these operational security requirements.

The first task is performed during view definition. It enables the DBMS to identify conflicts or inconsistencies among different views. A conflict of this kind occurs when one view designates a portion of the integrated database to be of its own exclusive use, while another view attempts to include that part of the database into its domain. These conflicts are not easily detected at the view translation level because, as will be explained later in this chapter, views are not made to communicate with each other at the view translation level. The view translation level performs mapping of constructs and translation of operations in each view independent of the existence of other views. On the other hand, constructs and operations are expressed in terms of the common conceptual data model when they reach the view enforcement level, therefore coordination can be facilitated here. This is graphically shown in fig 6.2.1.

The second task of this level is performed during database operation. All operations on a particular view, after being translated into operators on conceptual constructs by the view translation level, are checked against the security parameters maintained at this level.

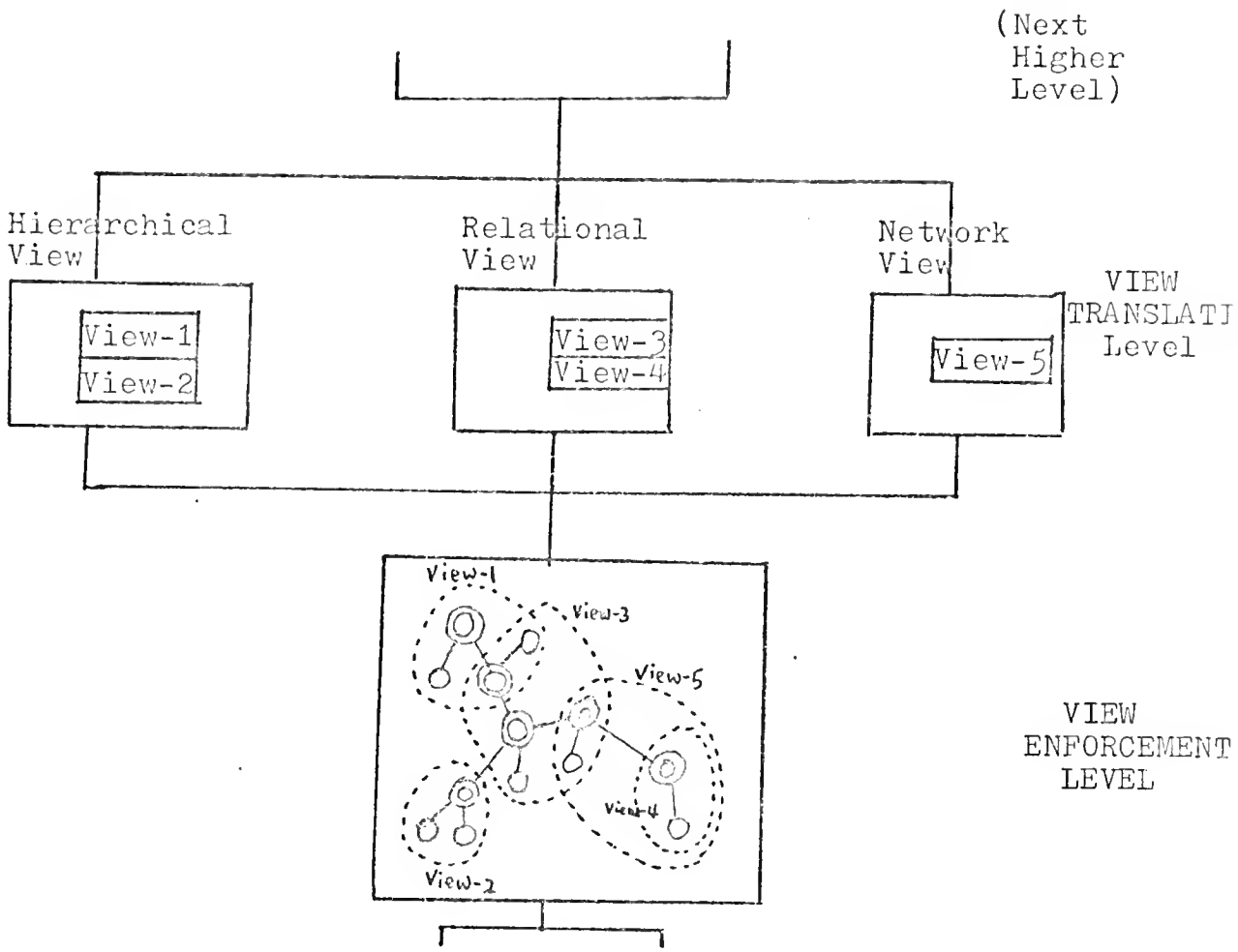


Fig 6.2.1 View Enforcement Level integrates and coordinates views

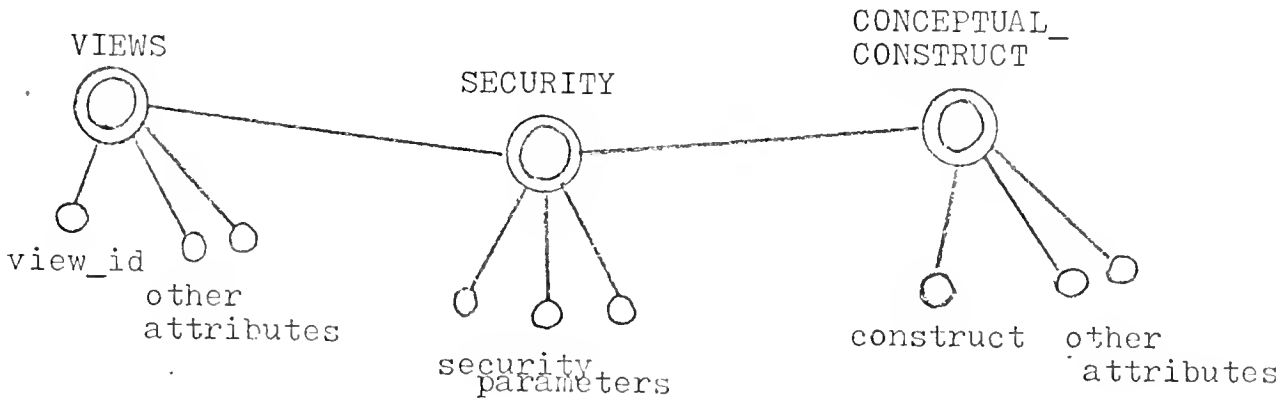


Fig 6.2.2: Catalogue structure at View Enforcement Level

6.2.2 General Mechanism

This level accomplishes its tasks by maintaining a catalogue, which contains information on views and conceptual constructs. It lists, for each conceptual construct (e.g., entities and attributes), the views that are allowed to read, write, share or exclusively use the construct. This enables the current level to identify conflicts between views and generate messages to effect intervention by a data base administrator to resolve the conflict. On the other hand, this information is also used during database operation to prevent a user from issuing operations not allowed within the view he is using.

This catalogue itself can be defined as entities and attributes. Two basic entities are VIEWS and CONCEPTUAL_CONSTRUCT. A third entity, called SECURITY, may be used to designate the many-to-many relationship between them. The binary network model of this catalogue is shown in Fig 6.2.2. This strategy of catalogue implementation makes use of functions provided at lower levels and releases the burden of catalogue maintenance from the view enforcement level.

6.2.3 Data Definition Interface

There are two parts in this interface. One is the conceptual schema definitions, whereby the view enforcement level obtains all construct names in the conceptual schema. More complicated parameters embodied in these conceptual schema definitions (such as virtual information and other semantic parameters) are of no concern to the

current level, and are passed down intact. The other part of this interface is the view definition, which consists of identification of a view and the corresponding conceptual schema this view is mapped to, as well as security requirements. The current level builds its catalogue using this information. The data definition interface is shown below:

```
Define_Nset (Nset_name, attr_list)
Define_View (View_id, conceptual_construct_list)
```

where the conceptual_construct_list is a list of conceptual constructs the view is mapped to and their corresponding operational security parameters.

6.2.4 Operational Interface

Operators to manipulate conceptual data elements are passed through. The operators are identical to those to be accepted by the next lower level, except for a tag which identifies the view based on which this operation is issued. Security checking is performed to enforce legality of this operation, and unauthorized operations are denied.

6.3 View Translation Level

This level incorporates several parallel modules, called external data model processors, each designed to handle a particular type of views, or external data model. In this section we describe mapping and translation from three data models, relational, hierarchical, and network, to the common conceptual data model. We shall illustrate how the mapping language introduced in section 6.1.1 is used to map constructs in these external data models, and how operators in these models are translated.

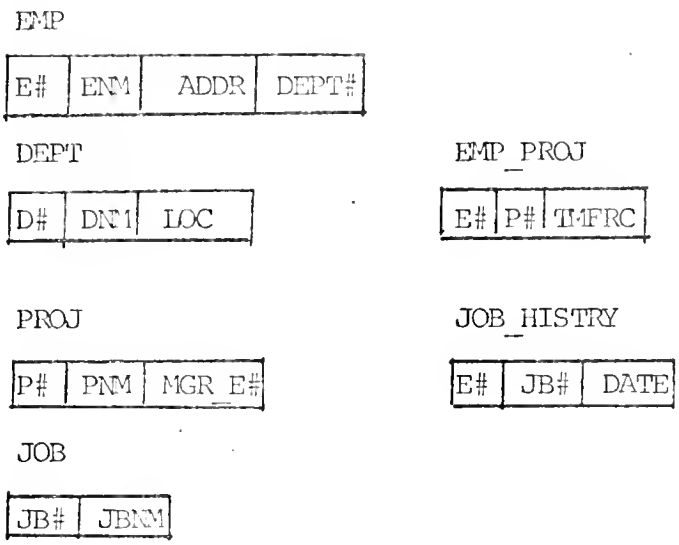
6.3.1 View Translation Level -- Relational View

6.3.1.1 Definition and Mapping

Relational views are defined in terms of relation names, domain names, attribute names and sequence attribute(s) (i.e. the attribute according to which the relation is to be sorted, if any). The mapping of these names to the conceptual constructs are also specified. Data types may be declared to be different from the form physically stored. The relation name gives rise to an entry in the relation catalogue, where information about this relation and its mapping is stored.

Since each attribute in a relation has to be atomic, an attribute name has to be mapped to a value set. The relation definition and mapping may take a format exemplified in Fig 6.3.1a, and Fig 6.3.1b shows the tree form of the mapping. Due to the great similarities between the relational data model and our conceptual n-ary entity sets,

Relational View



Relation definition and mapping

```
Define Relation EMP(E#,ENM,ADDR DEPT#)
                =EMP(E#,EN,ADDR,DEPT(D#))

Define Relation DEPT(D#,DNM,LOC)
                =DEPT(D#,DN,LOC)

Define Relation PROJ (P#,PNM, MGR_E#)
                = PRJ (P#,P_, MGR(E#))

Define Relation EMP_PROJ (E#, P#,TMFRC)
                = E_P (EMP(E#), PRJ(P#),TIMEFRAC)

Define Relation JOB (JB#, JBNM)
                = JOB(JB#,JBN)
```

Fig 6.3.1a A relational view of the EMP_ASSIGNMENT database

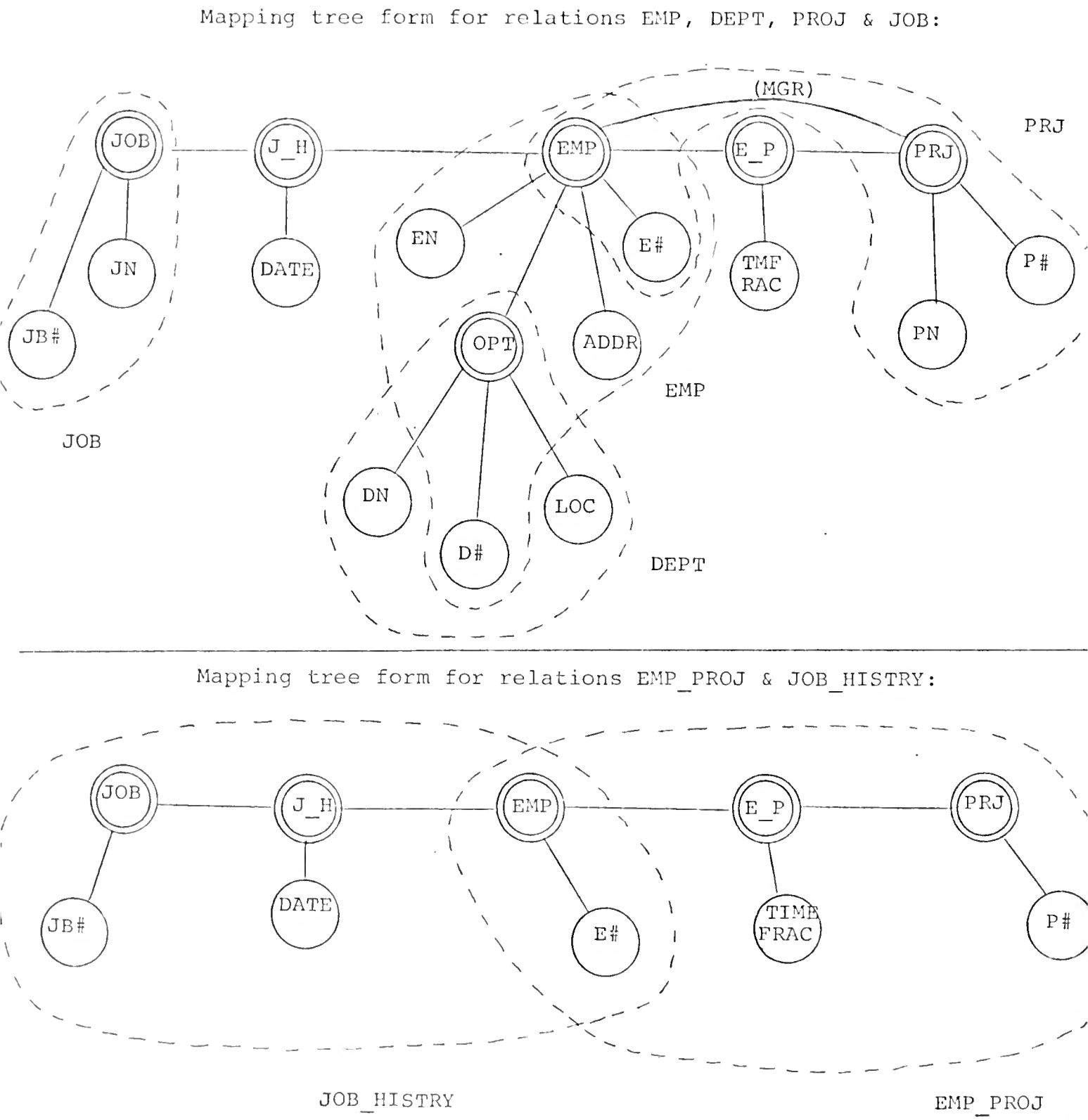


Fig 6.3.1b: Tree form of the relational view of EMP_ASSIGNMENT

mapping and translation of operators are fairly straightforward.

6.3.1.2 Tuple Construction

In our model, actual construction of the tuples does not take place when the relation is defined; only the mapping is stored with the relation name. Tuples of a relation are built one by one when an output command of the relation is encountered. It is built by passing n-ary entity retrieval commands extracted from the mapping definitions down to next levels. Because no relations are built when they are defined, redundancy of data is completely eliminated. Again, there are no restrictions on the normality of the relations.

6.3.1.3 Operations

Relational operator JOIN, SELECT, SELECT_WHERE, INSERT_TUPLE, DELETE_TUPLE, and UPDATE_TUPLE are supported. Get_next_tuple is also a command available for examining tuples one by one. This section describes the general logic how these operators may be translated and performed.

JOIN, SELECT and SELECT_WHERE commands are set operators that give rise to new relations. These relations, as usual, are not constructed, but their mappings are generated automatically and then stored. In the following discussions, the effect of these operations is exemplified by the tree form of the map.

JOIN operation involves joining two relations over a common domain. An Example is given in Fig 6.3.2.

SELECT operation would result in a 'pruned' tree, as shown in Fig 6.3.3. Note that selection that incorporates lower level roles but eliminates some intermediate level role would result in the latter being marked in the tree, but not eliminated. The reason for this is that, if we eliminate the intermediate level from the tree, the semantics of this relation may become ambiguous. In this example, if a command SELECT (R, (EMP_NO, LOC)) is given, the resulting relation shall have a form as shown in Fig 6.3.3b, while the tree shown in Fig 6.3.3c has an ambiguous semantics between the node EMP-NO and DEPT.

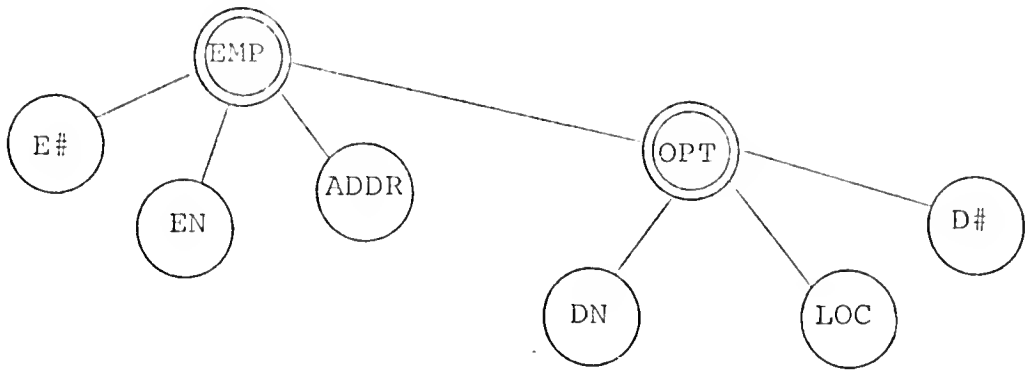
SELECT_WHERE is used to impose restrictions on values of certain attributes in a relation; this command is easily implemented by incorporating this restriction into the mapping tree. It is readily translated into Retrieve commands with WHERE clause implemented at the n-ary level. An example is given in Fig 6.3.4.

Due to semantic considerations, the update commands are restricted to normalized relations. However, if this restriction is lifted, other measures may be taken to remedy the semantic ambiguity.

INSERT_TUPLE: Inserting a tuple into an n-ary relation is equivalent to inserting a record into an nary entity sets. This command is translated into Insert_entity or Insert_attribute, depending on the context.

DELETE_TUPLE: Deleting a tuple from an n-ary relation is treated

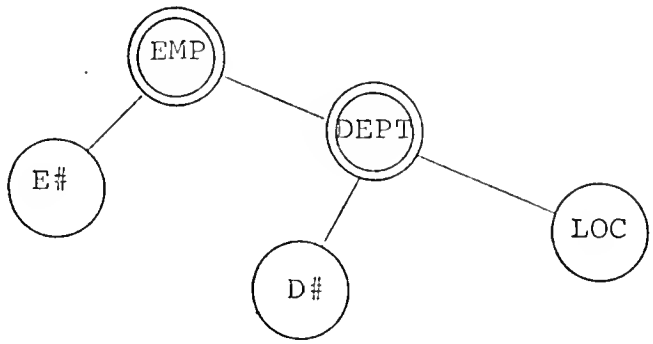
R = JOIN(EMP,DEPT,OVERD#)



∴ R(E#,ENM,ADDR,D#,DNM,LOC) = EMP(E#,EN,ADDR,DEPT(D#,DN,LOC))

Fig 6.3.2: Join operation results in a joined tree

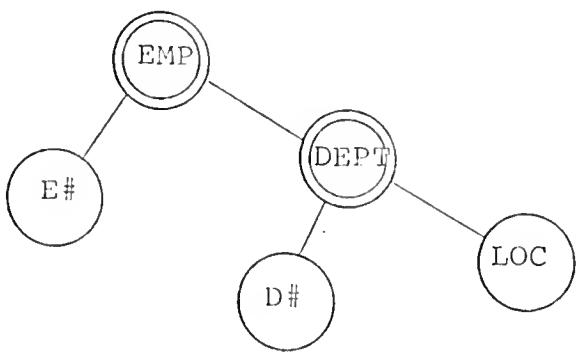
R1 = SELECT(R,(E#,D#,LOC))



∴ R1(E#,D#,LOC) = EMP(E#,DEPT(D#,LOC))

Fig 6.3.3a: Select operation results in a 'pruned' tree

R2 = SELECT(R1,(E#,LOC))



∴ R2 = N.EMP(E#,DEPT(LOC))

6.3.3b: D3 is marked not to be output

R2 = SELECT(R1,(E#,LOC))

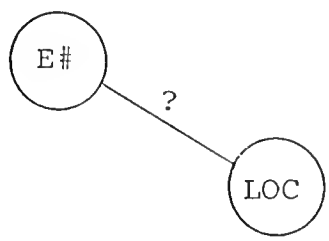
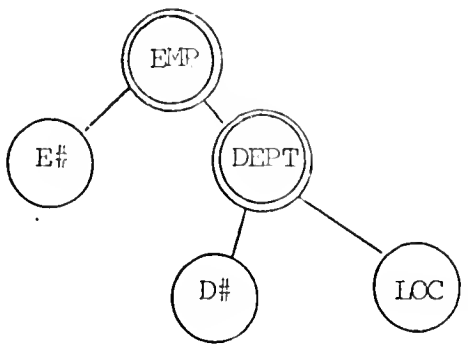


Fig 6.3.3c: Ambiguous tree map

```
R3 = SELECT (R, (E#, D#, LOC)) WHERE
     (LOC = "NEW YORK")
```




```
* R3(E#, D#, LOC) = EMP (E#, DEPT(D#, LOC))
                    WHERE (DEPT(LOC) = "NEW YORK")
```

Fig 6.3.4: SELECT WHERE operation

as deleting the entity designated by the key of this relation from the corresponding entity set. (It is clear at this point why a normalized relation should be referenced. Suppose that the relation is not normalized. Then one might ask the question: "what exactly does this user want to delete from the database?" Note that, in a tuple where several entities and binary relations are involved, removal of any of these would result in removal of the tuple.) This command is translated into Delete_entity or Delete_attribute, according to the context.

Update_Tuple: This operation may be translated into a sequence of Update commands to be passed down to the n-ary level.

6.3.1.4 Defining Relations Using Relational Operator

New relational views may be defined by commands JOIN, SELECT and SELECT_WHERE. A relation generated by these commands would be considered a temporary one, therefore not entered into the permanent relation catalogue unless attempt to save it is made. Once a relation is saved, it may be treated as a view, and other users may be granted access to it by calling the relation name and satisfying the access constraints. Therefore two commands are also available at this level to dynamically add entries to or remove them from the catalogue of relations:

Save_Relation Rel_name, access constraint parameteres

Delete_Relation Rel_name

where access parameters are extracted and processed at the next higher level, namely, the view authorization level.

6.3.1.5 Relational Sublanguage

The relational operators just described are used to manipulate relations in our model. However, the end users may still find them cumbersome to use, and query or manipulating languages of an even higher level, such as SEQUEL <Astrahan76>, may be desired. The database sublanguage facility, which will be described in section 6.4, provides translators to facilitate ease of user interface.

Operations other than those described here may be added (e.g., an operator that tests to see if two relations are equal, etc.). Due to the modularity of the system, the current repertoire of operators may be easily expanded to accommodate future needs.

6.3.2 View Translation Level -- Hierarchical View

6.3.2.1 Definition and Mapping

This type of external structure may be treated in a similar way as the relational structure. When the data definition and mapping of a hierarchy of segments are received, they are translated and stored in a catalogue. To illustrate, we again adopt the EMP_ASSIGNMENT database example shown in Fig 6.1. Now suppose a hierarchical view of this database, as shown in Fig 6.3.5, is to be generated. The definition of the view, which is very similar to an IMS type of DDL <IMSa>, with the addition of mapping specifications, may look like what is shown in Fig 6.3.6. Taking these definition statements as input, the DD translator may translate them into a set of parameters to be stored in the hierarchical view catalogue. A tree form of this hierarchy is shown in Fig 6.3.7.

6.3.2.2 Basic construct At Work

Comparing Fig 6.3.7 with tree forms demonstrated when we were discussing relational views, one may see how underlying binary relations may be used freely in constructing views. In essence, primitive and binary sets serve as the basic construct of the database, which are flexible enough to be built into any kind of external expressions, and has the property of relative stability over time. Another benefit that may be read from these figures is the clear

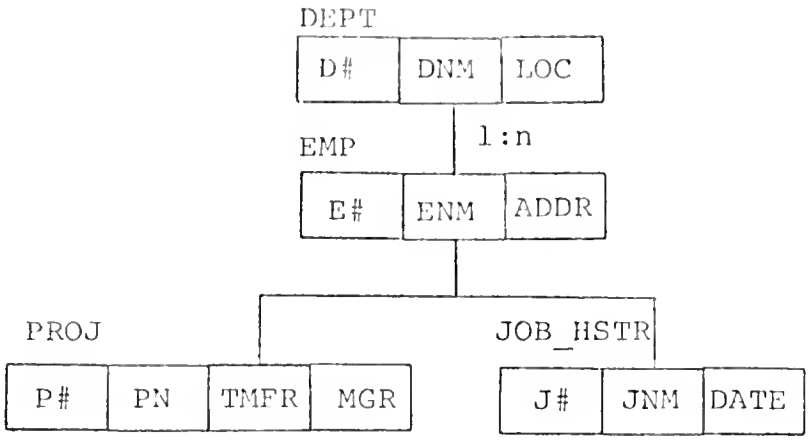


Fig 6.3.5: A Hierarchical view of EMP_ASSIGNMENT

```
HIERARCHY  NAME = EMP_ASSIGNMENT;
SEG        NAME = DEPT, BYTES = 45,
FIELD      NAME = D#, CHAR 5, SEQ,
FIELD      NAME = DNM, CHAR 20,
FIELD      NAME = LOC, CHAR 20;
       SEG.DEPT = DEPT(D#,DN,LOC) SEQ(D#);
SEG        NAME = EMP, BYTES = 45, PARENT = DEPT,
FIELD      NAME = E#, CHAR 5, SEQ,
FIELD      NAME = ENM, CHAR 20,
FIELD      NAME = ADDR, CHAR 20,
PARENT(DEPT)—— CHILD(EMP) = DEPT(EMP)
       SEG.EMP = EMP(E#,EN,ADDR) SEQ(E#);
SEG        NAME = PROJ, BYTES = 33, PARENT = SEG.DEPT
FIELD      NAME = P#, CHAR 5, SEQ,
FIELD      NAME = PN, CHAR 20,
FIELD      NAME = MGR, CHAR 5,
FIELD      NAME = TMFR, I3
PARENT(EMP)—— CHILD(PROJ) = EMP(PROJ)
       SEG.PROJ = E_P(PRJ(P#,PN,MGR(E#)),TIMEFRAC)
           SEQ(P#)
SEG        NAME = JOB_HSTR, BYTES = 31, PARENT = SEG.EMP
FIELD      NAME = J#, CHAR 5, SEQ,
FIELD      NAME = JN, CHAR 20,
FIELD      NAME = DATE, I6,
PARENT(EMP)—— CHILD(JOB_HSTR) = EMP(J_H)
SEG.JOB_HSTR = J_H(JOB(JB#,JBN),DATE) SEQ(JB#)
```

Fig 6.3.7: A tree form of Hierarchical view

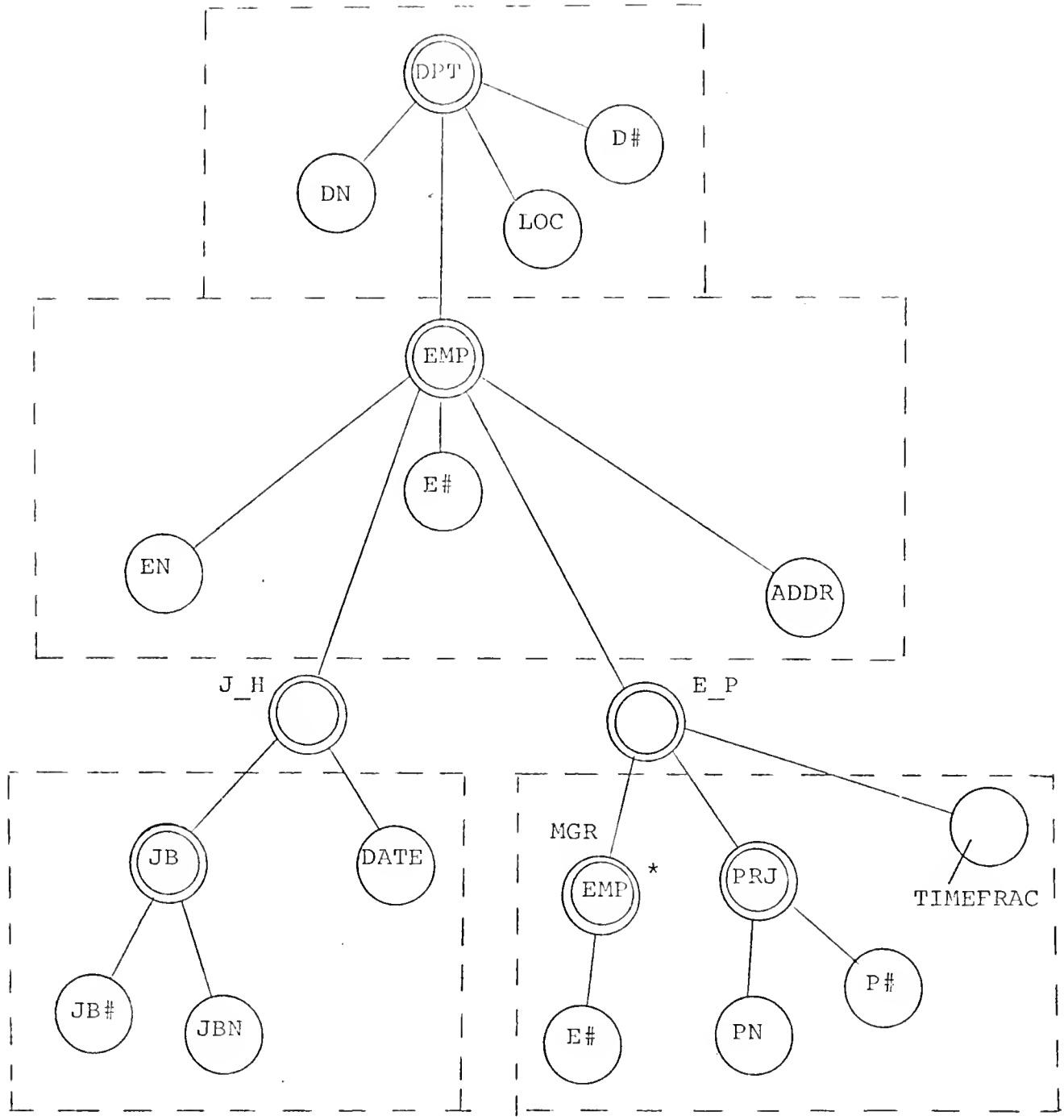


Fig 6.3.7: A tree form of Hierarchical view

semantics in an external view derived from binary associations. Not only may it help the DBA in clarifying the path during data definition, it also provides better documentation of the meaning of a database.

6.3.2.3 Operations

Conventional hierarchical operators GET_UNIQUE, GET_NEXT GET_NEXT_WITHIN_PARENT, DELETE, INSERT and REPLACE are supported. <IMSb>.

(a) GET commands

GET commands invoke the segment construction module which, by looking at the map and the content of the current buffer, determines how various data elements are to be retrieved and placed into output buffer. For example, a typical IMS retrieval command against our example database:

```
GU DEPT(DNAME='system')
   EMP
   JOB_HISTRY(JOB_NAME='programmer')
```

may be translated into operations on the nary entity sets, as shown in Fig 6.3.8.

(b) Update commands

```
Retrieve_unique (J_H (JOB (JB#,JBN), DATE))WHERE
                (EMP(DEPT(DN="SYSTEM"))) AND (JOB (JN ="PROGRAMMER"))
                SEQ (EMP(DEPT(D#),E#))
```

The tree form of a strategy to satisfy this request is:

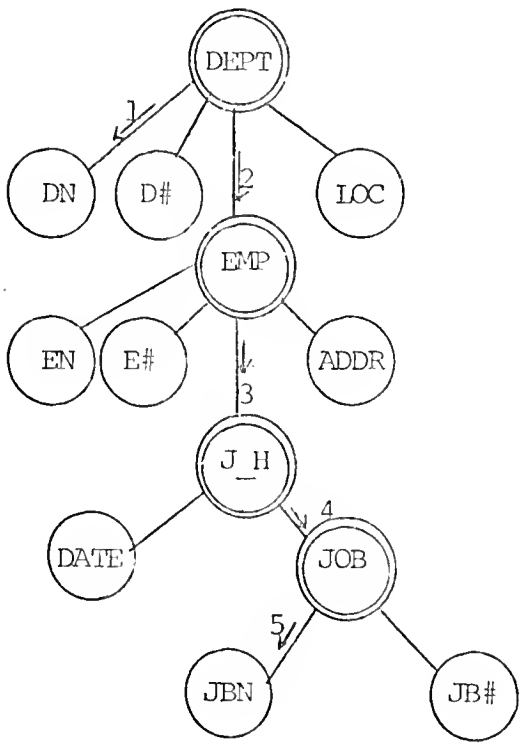


Fig 6.3.8: GU operation: tree form of the retrieval

Insertion of a segment is broken down into compatible Insert_entity and Insert_attribute commands. However, semantics of DELETE and REPLACE operations may need to be clarified in order to ensure proper update of the database. These issues are, again, similar to those discussed under relational views, and require special attention during design of data definition and manipulation languages.

(c) "Subschema"

A subset of a hierarchy of segments may also be defined to generate various views on this hierarchy. DDL is input to specify the name of the new 'subview' to be defined, as well as its connection to the original hierarchy. Access information is also given. Then this 'subview' is entered into the catalogue and may be operated upon by legitimate users.

6.3.3 View Translation Level -- Network View

6.3.3.1 Definition and Mapping

In this section we discuss definition and mapping of network views <DBTG71>. Based on the conceptual definition of the database EMP_ASSIGNMENT of Fig 6.1, the system would like to present to the user a DBTG type of network view as shown in Fig 6.3.9a. The definition and mapping language is shown in Fig 6.3.9b. In essence, the records defined in the network are mapped to entity sets, their identifiers mapped to key attributes of the entities, and DBTG 'sets' are mapped to underlying binary associations. A tree form of this mapping is shown in Fig 6.3.9c.

6.3.3.2 Operations

Operations on a network view include <Date77>:

Find: Retrieves any record or a specific record within a set

Modify: Writes content of a record back to the database

Insert: Insert a record into a set

Remove: Removes a record from a set

Delete: Deletes a record from the database

Store: Inserts a record into the database

The 'currency' concept used in DBTG model is utilized in the tranlation of these commands. The user is given a UWA (User Working Area),

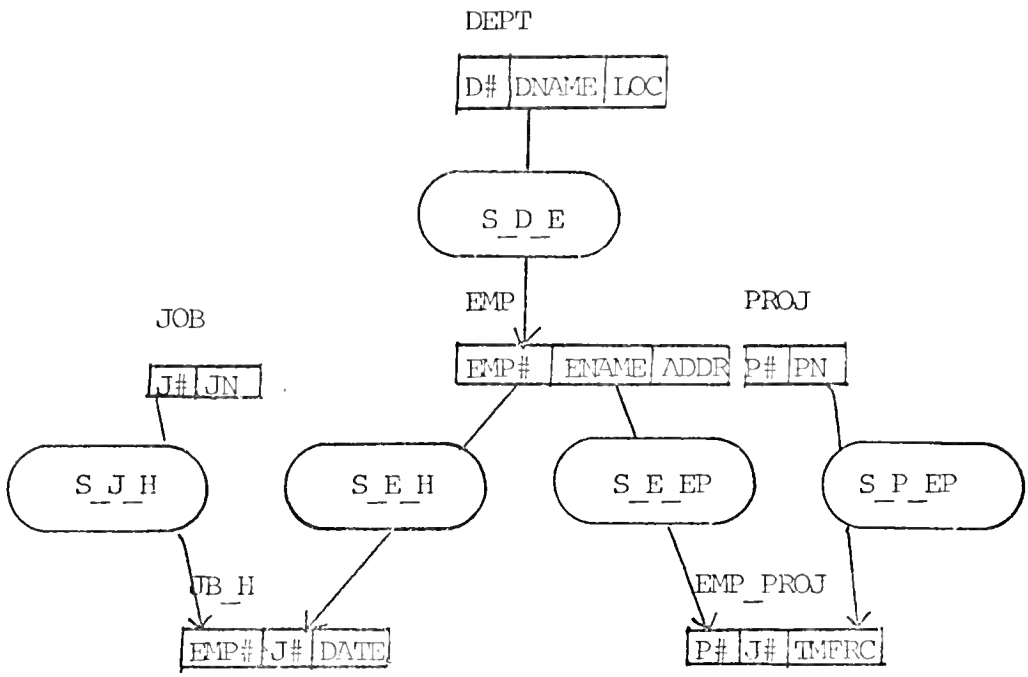
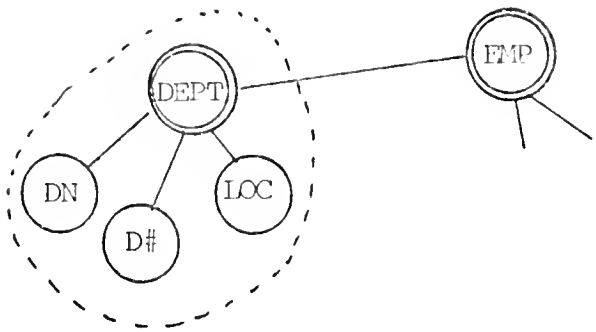


Fig 6.3.9a: A network view of the EMP_ASSIGNMENT database

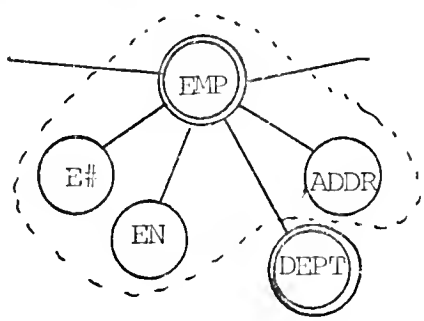
```
  NETWORK  EMP_ASSIGNMENT

  RECORD   DEPT, IDENT IS DEPT# IN DEPT

           02 DEPT#, CHAR5
           02 DNAME, CHAR20
           02 LOC, CHAR20

* REC.DEPT = DEPT(D#,DN,LOC);

  RECORD   EMP, IDENT IS EMP# IN EMP

           02 EMP#, CHAR5
           02 ENAME, CHAR20
           02 ADDR, CHAR30

* REC.EMP = EMP(E#,EN,ADDR);

  RECORD   JOB, IDENT IS J# IN JOB

           02 J#, CHAR5
           02 JN, CHAR20

* REC.JOB = JOB(JB#,JBN);

  RECORD   JB_H, IDENT IS J# IN JOB, EMP# IN EMP

           02 J#, CHAR 5
           02 EMP#, CHAR5
           02 DATE, CHAR5

* REC.JB_H =  J_H (EMP(E#), JOB(JB#), DATE)

  SET      S_D_E, OWNER IS DEPT, MEMBER IS EMP

* SET(S_D_E) O → M = DEPT(EMP)
             M → O = EMP(DEPT)
```

Fig 6.3.9b: An example of data definition and mapping of a network view

Treeform of DEPT record:



Treeform of EMP record:



Treeform of S_D_E set:

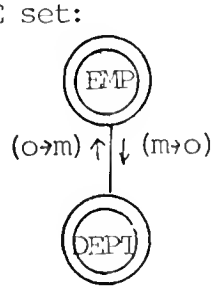


Fig 6.3.9c: Treeform of a newtork view of EMP_ASSIGNMENT

containing a buffer for every record type defined in the view. The content of this UWA, together with the mapping definition, is used to generate retrieval/update commands against the nary entity level. To illustrate, suppose that the network and mapping definitions are stored in a data structure exemplified by Fig 6.3.10. During run time, a UWA buffer is reserved as shown in Fig 6.3.11; and the basic logic together with some examples of translation of the DBTG commands is shown in Fig 6.3.12.

MAP1:

Rec_name	NSET	Key_attr
DEPT	DEPT(D#,DN)	D#
JB_H	J_H(EMP(E#),JOB(JB#), DATE)	(EMP(E#),JOB(JB#))

MAP2:

Set_name	OWNER_REC	MEM_TO_OWNER_ATTR	MEM_REC	OWNER_TO_MEM_ATTR
S_D_E	DEPT	DEPT	EMP	EMP

Fig 6.3.10: An example format of data structure of the network database mapping

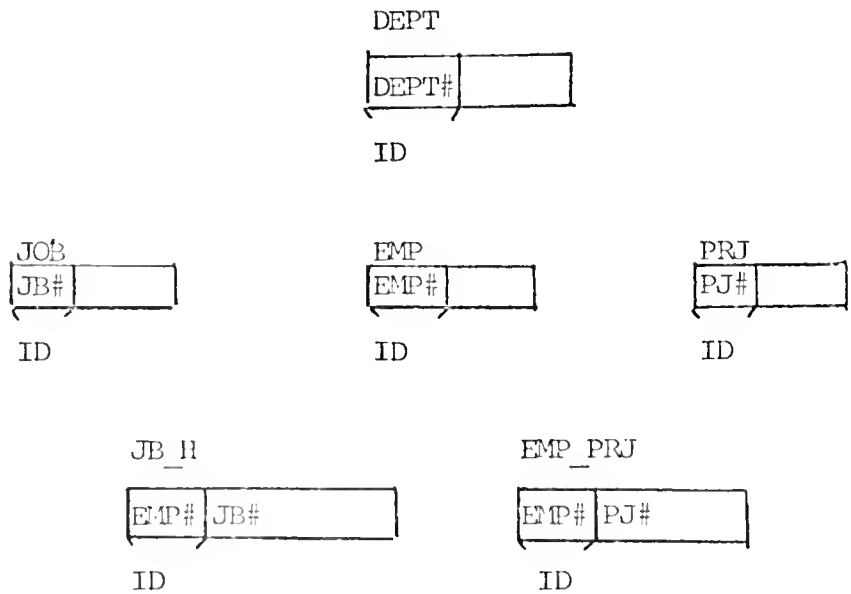


Fig 6.3.11: UWA of EMP_ASSIGNMENT network

```
FIND Rec_name :

      Retrieve_unique (NSET(Rec_name))where
                      (Key_attr(Rec_name)=ID(Rec_name))

      (Here NSET(Rec_name) refers to the second column of MPA1 in
       Fig 6.11; Key_attr(Rec_name) refers to the 3rd column of MAP1;
       ID(Rec_name) refers to the ID field of the record Rec_name
       in UWA; etc.)

FIND FIST(NEXT) MEMBER WITHIN Set_name:

      Retrieve_first(next) (NSET(MEM_REC(Set_name)) where
                           (MEM_TO_OWNER_ATTR(Set_name) =
                            ID(OWNER_REC(Set_name)))

FIND OWNER WITHIN Set_name:

      Retrieve_unique (NSET(OWNER_REC(Set_name))) where
                      (OWNER_TO_MEM_ATTR(Set_name) = ID(MEM_REC(Set_name)))
```

Fig 6.3.12a: General logic for translating FIND commands

```
    MOVE "15" TO DEPT# IN DEPT;
    FIND DEPT;
*   RETRIEVE_UNIQUE (DEPT (D#,DN,LOC)) WHERE (D#= DEPT# in DEPT in UWA);
    DNAME IN DEPT = "SYSTEM";
    MODIFY DEPT;
*   UPDATE (DEPT) (D#=DEPT# in DEPT in UWA) (DN= DNAME in DEPT in UWA,
                   LOC=LOC in DEPT in UWA);
    FIND FIRST EMP WITHIN S_D_E;
*   RETRIEVE_UNIQUE (EMP (E#,EN)) WHERE(DEPT(D#)= DEPT# in DEPT in UWA);
    FIND FIRST EMP_PROJ WITHIN S_E_EP;
*   RETRIEVE_UNIQUE (E_P (EMP(E#),PRJ(P#),TIMEFRAC) WHERE
                     (EMP(E#)=EMP# in EMP in UWA);
    FIND OWNER WITHIN S_P_PJ;
*   RETRIEVE_UNIQUE (PRJ (P#,PN)) WHERE (P#= P# in EMP_PRJ in UWA);
    PRINT PROJ;
    FIND NEXT EMP WITHIN S_D_E;
*   RETRIEVE_NEXT (EMP (E#,EN)) WHERE (DEPT(D#)=DEPT# in DEPT in UWA)
                   SEQ(E#) CURRENT IS (EMP# in EMP in UWA);
```

Fig 6.3.12b Example of trnaslation of a set of network data model commands
(* denctes translated statements)

6.3.4 Database Sublanguage Facility and Summary of View Translation Level

As mentioned in section 6.3.1, a user of the relational data model may wish to use a relational sublanguage such as SEQUEL to query the database. In general, any very-high-level user-oriented database sublanguage may be incorporated into our system, so long as the data model it assumes is supported at the view translation level and operators it uses can be mapped to the operators of that particular data model. A graphical representation of this concept is shown in Fig 6.3.13. Note that some sublanguages may be built on top of different external data models simultaneously, depending on the application it supports.

In summary, the three types of views discussed are very different from each other in terms of definitions and the set of operations allowed. However, implementation of these views on top of our conceptual sets is similar, since all of them involved breaking the views into the basic constructs. Therefore the final mapping may have a common format.

In general, other views may also be provided, so long as they can be broken down into a clearly defined collection of basic constructs supported by the conceptual schema. Once this is done, any operation on the views may be translated.

Finally, we should take note of the issue of performance. Since we have truely established a system of many levels of indirection, care

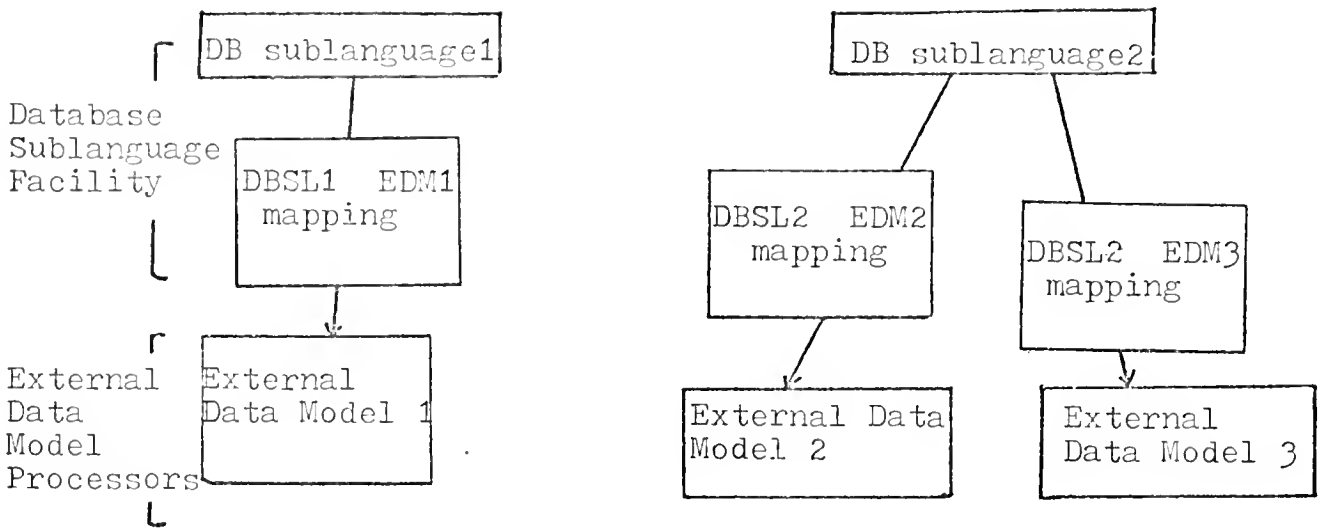


Fig 6.3.13 Architecture for handling
Database Sublanguages

has to be taken to address the problem of overhead. How we may proceed to take advantage of sequential processing by penetrating through these levels in order not to lose connectivity between one transaction and the next is one of the vital performance issues we shall look at when an external view and processings against it are established.

6.4 View Authorization Level

6.4.1 Introduction

Every external view (e.g. a set of relations or a hierarchy of segments) has associated with it a set of legal 'viewers', which is designated by a set of accounts. Legitimate user accounts of a view are specified when a view is defined.

Hierarchy of authority: The access power of accounts may be organized hierarchically; for example, accounts beginning at letter A may have all the access capabilities of accounts beginning at letter B, but not vice versa. Also, those accounts that are capable of defining views will have the authority to designate a set of accounts as legal users and specify their capability (e.g. read or write). On the other hand, a view definition may only be changed or deleted by an account that has the authority to define the view. Therefore a hierarchy of authorization is constituted.

Log-on: As conventionally done, a user gains access to an account through its password which is to be supplied to the system when he attempts to log on. Once the user is in the system, a process is created. The process controller at the front end level maintains information of all active processes.

View authorization: For the purpose of security control, this level maintains several control tables. The first one is an access

List. It lists, for each view defined for the database, the accounts that are allowed to retrieve or update the view. It also lists those accounts that have the authority to change the view definition. An example is given in Fig 6.4.1.

The other table is an active process table, which, for each active process in the system, lists the view currently in use by the process. A process has to declare the view it wishes to operate upon before any access command is given. Once the view is declared (through an OPEN_VIEW command), and the access list checked for legality of this declaration, an entry in the active process table is made. The format of the table is shown in Fig 6.4.2. The view authorization level also posts this information with the view translation level, which then proceeds to create a 'process control block' for this process within the appropriate external data model processor. This 'process control block' also maintains working space necessary for those view construction procedures to serve this process. A process may change the view it wants to see by closing the previous one (by CLOSE_VIEW command) and declaring a new one. A summary of the command flow at this level is shown in Fig 6.4.3.

6.4.2 DD Interface

Database designers have to spell out access information when the database or a view of it is defined. All authorization parameters in DD statements are intercepted and entered into the catalogue, and further manipulation on the database are checked against this security

VIEW NAME	READ	WRITE	DEFINITION
Hier_EMP	B001- B009	A001 A030	A900
Hier_DEPT	B010- B019	A020	A901

Fig 6.4.1 An example of the format of the access list

PROCESS ID	ACCOUNT	VIEW OPENED	CAPABILITY
B001.1	B001	Hier_EMP	Read
A001.1	A001	Hier_EMP	Write
B001.2	B001	Rel_DEPT	Read

Fig 6.4.2 Active process table

```
VAL: BEGIN;

     Case command Of

     "Define_View": Do;

             Process View Authorization definition;
             If definition legal then do;

                Make entry in the access list;
                Pass the rest of the data definition
                   to an appropriate EDMP;
                end;

             Else do;

                Formulate error message;
                Pass error message out;
                end;
             end;

    "Open_View": Do;

             Check against the acess list;
             If legal then do;

                Make entry in the active process table;
                Pass command to the appropriate EDMP;
                end;

             Else do;

                Formulate error message;
                Pass error message out;
                end;
             end;

   "Close_View": Do;

             Erase view entry in the active process table;
             end;

   "Operations": Do;

             Check against active process table:
             Pass command to the EDMP which processes the view;
             end;

   END;

   (EDMP: External Data Model Processor)
```

Fig 6.4.3: Command flow in the View Authorization Level

information in the catalogue.

A special module called authorization_parameter_processor is used to decode the security parameters and make entries in the security catalogue before passing the rest of the DD statement down.

6.4.3 Operational Interface

The set of operators accepted at this level for data manipulation purposes is identical to those accepted at external view levels. In addition, a process must open a view before it issues an operator against it, and close the view before it wishes to change to another view or becomes inactive. Therefore two additional commands are identified:

```
OPEN_VIEW (process_id, account_id, view_name)
CLOSE_VIEW (process_id)
```

VII. SUMMARY AND FUTURE RESEARCH DIRECTIONS

7.1 Summary of report

The INFOPLEX database computer project has provided the motivation for this report. It is our belief that, while information processing is and will be playing the central role in the application of computers, the conventional computing-oriented computer architecture is not adequate for handling large-scale information processing. The INFOPLEX database computer is geared toward design of a highly-parallel computer system specialized in information processing.

In chapter 1, the general background and basic architectural concepts of the INFOPLEX database computer are introduced. The INFOPLEX consists of two parts, the Storage Hierarchy and the Functional Hierarchy. The Storage Hierarchy handles an ensemble of storage devices and supervises data movement among them, supporting a large virtual storage; while the Functional Hierarchy performs all other database management functions. This report presents a preliminary design of the latter component based on the concepts of hierarchical functional decomposition and multiple microprocessor implementation.

The first task during preliminary design is to identify the functional requirements of the system. In chapter 2, a search into the literature of database systems has helped clarifying the picture. Most notably, two concepts in stratification of database managment systems,

namely, information abstracton and functional abstraction, are reviewed to provide insights into functions to be supported by a contemporary database system and how to organize them into hierarchical level. The following are important functional objectives of the functional hierarchy: (1) multiple types of external views; (2) a high-level conceptual data model; (3) a variety of stored data structures (i.e. a flexible internal model); (4) explicit support of security, validity, alerting and virtual information; and (5) concurrent use of the database.

A Binary Network data model is developed as the conceptual data model in the functional hierarchy. The model is shown to provide natural mapping to the external views and the stored data structure, and incorporates clean semantics. The internal data model and external view support are also outlined. Chapter 2 concludes with a 10-level hierarchy of functions, which are further detailed in later chapters. These functions may be grouped into memory management, internal structure management, conceptual structure management, and external structure management.

The memory management level uses the id approach to insulate the byte detail from the upper levels, and keeps track of free storage space and performs compaction and garbage collection. The id approach enables all upper levels to use an id as the location of a data item, and not to be concerned about the byte address. The internal structure management is broken into three levels, integrated by the Basic Encoding Unit (BEU) concept. The data encoding level provides 'final touches', such as text editing, compaction and encryption, to the data

before the data enters into the storage hierarchy. The unary set level organizes data elements into unary sets, and is capable of performing intellignent search into a unary set for a particular unary data element. The binary association level maps binary relations described in the conceptual schema to their implementation, and pieces together unary data elements (records) and pointers among them. These three internal structure levels provide insulation between the conceptual organization of information and its internal structure such that changes in internal structures will be reflected only in the mapping between the two. The interface to the binary association level enables upper levels to dedicate search and storage of their own house-keeping information (e.g. catalogues) to the internal levels, thus realizing functional abstraction. Furthermore, the BEU concept reflects a parametric and modular approach to internal structure building.

The conceptual structure management is also broken into 3 levels. The n-ary level processes the core part of the conceptual schema, keeping track of entities and attributes and enforces binary semantic relationships among them. The virtual information level builds new consructs whose values are not stored, but derived. The validity level maintains more complex update constraints.

The external structure management provides user views onto the integrated database. Three types of views are demonstrated to be mappable to the conceptual structure. The view translation level keeps track of construct mapping and performs operator translation. Finally, system security is maintained through a view mechanism contained in the view enforcement level and the view authorization level.

To summarize, processing of a request in the database is described below. It is first checked for entry legality by the view authorization level; then it enters into the view translation level, which translates its references to external constructs into those in the conceptual schema. The request then goes into conceptual structure levels, which in turn call up internal structure levels to retrieve or update the target elements. Each level in performing its task may call upon lower levels for information or subtasks. The system employs both transaction pipelining and functional abstraction. It also supports the notion of 'family of systems' by having levels communicate with others through implementation-independent interfaces, allowing easy reconfiguration of the system.

7.2 Future research directions

We plan to conduct further research in the area of the functional hierarchy along the following dimensions.

7.2.1 Formal Design Methodology

In this report, we have identified functional requirements of a database system and organized them into a hierarchical structure. A formal design methodology will be utilized such that the current design can be verified and alternative designs examined. A formal design methodology will also help in detecting potential ambiguities in the design before implementing a software prototype of the system. We plan

to build our methodology based on the Systematic Design Methodology (SDM) developed by Huff <Huff79>. Some extensions of SDM may have to be generated to tie this design methodology more closely to the design of the functional hierarchy. This research will include both investigation and application of the methodology.

7.2.2 Locking Mechanisms

To support concurrent uses of a shared database, interlock mechanisms must be used to coordinate update operations. Care has to be taken in designing and implementing this locking mechanism to avoid adversely affecting performance of the system. Current research in this area includes its theoretical aspects <e.g., Bernstein80b, Eswaran74>, strategies used for concurrency control in database systems <e.g., Gray76, Bernstein80a>, and certain performance issues <e.g., Badal80, Ries77>. They will be reviewed in an attempt to develop a method that is most suited to the architecture of the functional hierarchy. The relationship between concurrency control at the functional level and at the physical level will also be investigated.

7.2.3 Mapping of the operators

The need for mapping stems from the fact that the functional hierarchy is composed of layers each of which supports a different set of operators. The differences in data structures and data models between levels contribute most to the differences in these operators.

Further research is required to

1) specify in more detail the meaning of the operators at each level;
2) show how these operators are translated into those implemented at the next lower level;
3) prove that the translation algorithms preserve the desired meaning of the operators.

Current-day research in the area of data models, much of it having been reviewed in chapter two (notably section 2.2.3), will be drawn upon to obtain insights into this problem.

7.2.4 Implementation of a software prototype

For better understanding of the functional hierarchy, implementation of a software prototype shall be planned. One of the major purposes of this software prototype is to facilitate measurements of parameters for performance evaluation. The following steps will be taken for carrying out this implementation:

1) Select a member system from the family of the database systems supported by the functional hierarchy;
2) Resolve the detailed design issues and algorithms in the selected system;
3) Conduct coding and testing of the system in PL/1;
4) Identify properties in performance of the system.

7.2.5 Performance evaluation

Models will be built to examine performance of the proposed architecture. Measurements will be taken from the software prototype discussed above as estimates of some of the parameters in the model. Experiments will be conducted to closely examine performance of the system in various user environments and implementation alternatives. A comparison in performance will also be made between the proposed, highly parallel and decentralized computer architecture and the conventional architectures.

7.2.6 Recovery and reliability

One of the advantages of the proposed architecture of the functional hierarchy is its ability to recover from isolated software and hardware breakdowns. Further research is required to identify the methods and protocols to be used by the functional hierarchy to detect and recover itself from possible breakdowns. Measurements of reliability are to be taken for comparison with the conventional architecture.

REFERENCES

Abraham79: Abraham, M., 'Properties of reference algorithms in multilevel storage hierarchy' Master Thesis, Sloan School of Management, MIT, 1979

Astrahan76: Astrahan, M.M. et al. 'System R' ACM TODS, Vol 1, NO. 2, June 1976

ANSI75: ANSI/X3/SPARC study group on DBMS interim report, Feburary 1975

Bachman77: Bachman, C.W., and Daya, M., 'The role concept in data models', Proc., VLDB, 1977

Badal80: Badal, D.Z., 'The analysis of the effects of concurrency control on distributed database system performance', VLDB 1980

Bernstein76: Bernstein, P.A., 'Synthesizing third normal form relations from functional dependency', ACM TODS, Vol. 1, No. 4, Dec 1976

Bernstein80a: Bernstein, P.A., Shipman, D.W., and Rothnie, J.B., 'Concurrency control in a system for distributed databases (SDD-1)' ACM TODS Vol. 5, No. 1, March 1980

Bernstein80b: Bernstein, P.A., and Goodman, N., 'Fundamental Algorithms for concurrency control in distributed database systems', Technical Report CCA-80-05, Computer Corporation of America, Feb 1980

Borkin78: Borkin, S.A., 'Data model equivalence', Proc., VLDB, 1978

Bracchi74: Bracchi, G. et al. 'A multilevel relational model for database management systems', in Data Base Management, North-Holland Publishing Company, 1974

Bracchi76: Bracchi, G., Paolini, P., and Pelagatti, G., 'Binary logical associations in data modelling', in Modelling in Data Base Management System, (ed., Nijssen), North-Holland, 1976

Chen76: Chen, P. P., 'The entity-relationship model -- toward a unified view of data', ACM TODS Vol 1, No. 1, March 1976

Codd72: Codd, E.F., 'Further normalization of the database relational model', in DataBase Systems, Prentice-Hall, 1972

Codd79: Codd, E. F., 'Extending database relational model to capture more meaning' ACM TODS, vol 4, no. 4, December 1979

Date77: Date, C.J., 'An introduction to database systems' Addison-Wesley Publishing Company, 1977

DBTG76: Data Base Task Group of CODASAL Programming Language Committee, COBOL Journal of Development, 1976

Eswaran74: Eswaran,K.P., et.al., 'The notions of consistency and predicate locks in a database system', IBM RJ 1329, December 1974

Eswaran75: Eswaran,K.P. and Chamberlin, D. D., 'Functional specifications of a subsystem for database integrity', Procd., VLDB, 1975

Fagin77a: Fagin, R. 'Multivalued depencdencies and a new normal from for relational database', ACM TODS, Vol. 2, No. 3, September 1977

Fagin77b: Fain, R., 'The decomposition versus the synthetic approach to relational database design', Proc., VLDB, 1977

Falkenberg76: Falkenberg, E. 'Significations: the key to unify data base management', Information Systems, 1976, Vol 2, No.1

Falkenberg77: Falkenberg, E., 'Concepts for the coexistence approach to database management', Proc., Int. Comp. Symp., April 1977

FODS76: 'Proposal for research on the design of a family of data base systems' CISR Draft, Sloan School of Management, MIT, December 1976

FOS76: Madnick & Goldberg, 'Family of Operating Systems', Sloan School of Management, MIT, 1976

Fry77: Fry, J.P. et al., 'Stored-data description and data translation: a model and language', Information Systems, Vol 2, 1977, PP.95-148

Gray76: Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.L., 'Granularity of locks and degrees of consistency in a shared database' in Modelling in database management systems, North Holland Publishing Company, 1976

Hall76: Hall, P., Owlett, J., and Todd, S., 'Relations and Entities', in Modelling in Database Management System, North-Holland, 1976

Housel79: Housel, B.C., Waddle, V., and Yao, S.B., 'The functional dependency model for logical database design', Proc., VLDB, 1979

Hsiao77: Hsiao, D. K., Madnick, S.E. 'Database machine architecture in the context of information technology evolution', Proceeding, VLDB 1977

Hsiao79a: Hsiao, D.K., Kerr, D.S., Madnick, S.E., 'Computer security' Academic Press 1979

Hsiao79b: Hsiao, D.K., ed., 'Collected readings on a database computer (DBC)', Department of computer and Information Science, The Ohio State University, Columbus, March 1979

Huff79: Huff, S.L., 'A systematic methodology for designing the architecture of complex software systems', Ph.D. Thesis, Sloan School of Management, MIT, June 1980

IMSa: IBM Information Management System/ Virtual Storage Utilities Reference Manual

IMSb: IBM Information Management System/ Virtual Storage Application Programming Reference Manual

Kerschberg77: Kerschberg, L., Klug, A., and Tsichritzis, D., 'A taxonomy of data models', Proc., VLDB, 1976

Klug77: Klug, A. and Tsichritzis, D., 'Multiple view support within the ANSI/SPARC framework', Proc. VLDB, 1977

Lam79: Lam, C., Madnick, S.E., 'Properties of storage hierarchy systems with multiple page sizes and redundant data' ACM TODS, September, 1979

Madnick69: Madnick, S.E. and Alsop, J.W., 'A modular approach to file

system desing', Proc., AFPIS, 1969

Madnick73: Madnick,S.E. et al., 'Virtual Information in data base systems', Sloan School of Management, M.I.T.

Madnick74: Madnick, S.E., Donovan, J.J. 'Operating Systems', McGrqw-Hill, New York, 1974

Madnick75: Madnick, S.E., 'Design of a general hierarchical storage system', IEEE International Conference Procedings, 1975

Madnick79: Madnick, S.E., 'The INFOPLEX database computer: concepts & directions' Proceedings, IEEE Computer Conference, Feburary 1979

Madnick80: Madnick, S.E., 'Proposal for Research on the Desgin of a high-erformance high-availability intelligent memory system', Sloan School of Management, MIT, May 1980

McLeod78: McLeod, D. 'A semantic database model and its associated structured user interface', Ph.D. Thesis, L.C.S., MIT, August 1978

Morgan75: Morgan, H.L., and Buneman, O.P., 'Alerting in database systems: concepts and techniues', working paper 75-12-02, The Wharton School

Navathe76: Navathe, S.B., and Fry, J.P., 'Restructuring for large databases: Three levels of abstraction', ACM TODS, Vol 1, No. 2 June, 1976

Navathe77: Navathe, S.B., 'Schema analysis for database restructuring', VLDB Proceedings, 1977

Navathe78: Navathe, S.B. and Schkolnick, M., 'View representation in logical database design', Proc., ACM SIGMOD 1978

Nijssen76: Nijssen G.M. ed., Modelling in Data Base Management Systems, North-Holland Publishing Company, New York 1976

Paolini77: Paolini, P. and Pelagatti, G., 'Formal definitions of mappings in a database', Proc., ACM SIGMOD 1977

Ries77: Ries, D.R., and Stonebraker, M., 'Effects of locking granularity in a database management system', ACM TODS Vol.2, No.3, September 1977

Rothnie76: Rothnie, J.B. and Hardgrave, W.T., 'Data model theory: a beginning', Texas Conference on computer Systems, 1976

Roussop75: Roussopoulos, N. and Mylopoulos, J., 'Using semantic networks for data base managment', Proc., VLDB 1975

Schkolnick78: Schkolnick, M., 'A survey of physical database design methodology and techniques', IBM Research RJ 2306, August 1978

Schmid75: Schmid, H.A., Swenson, J.R., 'On the semantics of the relational data model', ACM SIGMOD Conference Proceedings, 1975

Senko73: Senko, M.E., 'Data structures and accessing in database systems: II. Information Organization', and 'III. Data representations and data independent accessing model', IBM Systems Journal, No.1 1973

Senko77: Senko, M.E, 'Data strucutre and data accessing in database systems: past, present and future', IBM Systems Journal No.3, 1977

Smith71: Smith, D.P., 'An approach to data description and conversion', Moor School Report No. 72-20, University of Pennsylvania, 1971

Smith77a: Smith J.M., Smith D.C.P., 'Database abstractions: Aggregation and generalizations', ACM TODS Vol 2, No.2, June 1977

Smith77b: Smith, J.M., and Smith, D.C.P, 'Database abstractions: Aggregation', CACM, Vol 20, No. 6, 1977

Toong80: Toong, H.D., 'A general multi-microprocessor interconnection mechanism for non-numeric processing', Fifth Workshop on Computer Architecture for non-numeric processing, March 1980

Tsichritz78: Tsichritzis D. et al. 'The ANSI/SPAC DBMS framework', Information Systems, 3,3,1978, pp.173-192

Vetter77: Vetter, M., 'Database design by applied data synthesis',

VLDB Proc., 1977

Yao79: Yao, S.B., 'Optimization of query evaluation algorithms', ACM TODS Vol 4, No. 2, June 1979

Yourmark77: Yormark, B. 'The ANSI/X3/SPARC/SGDBMS architecture' 1977